AD-A062 462 MASSACHUSETTS UNIV AMHERST F/G 12/2
HARDWARE ENHANCEMENT OF OPERATING SYSTEMS.(U)
NOV 78 C C FOSTER DAAG29-76-G-0335
UNCLASSIFIED ARO-14230.1-EL NL

1 OF 1
AD A062462

END
DATE FILMED
3-79
DDC

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ARO 14230.1-EL

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: None
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): Hardware Enhancement of Operating Systems.

5. TYPE OF REPORT & PERIOD COVERED: Final, Sept. 23, 1976 - Sept. 23, 1978

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): Caxton C. Foster

8. CONTRACT OR GRANT NUMBER(s): DAAG 29-76-G-0335

9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Massachusetts

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 93p.

11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, Post Office Box 12211, Research Triangle Park, NC 27709

12. REPORT DATE: Nov 23, 1978

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: NA

LEVEL II

16. DISTRIBUTION STATEMENT (of this Report)

Final rept. 23 Sep 76 - 23 Sep 78,

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

DAAG-29-76-G-0335

18. SUPPLEMENTARY NOTES

ARO 14230.1-EL

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Operating Systems, Hardware enhancement, Content Addressable memories.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This study investigated what hardware could be added to a conventional computer in order to speed up the execution, decrease the complexity and improve the reliability of an operating system. It was found that content addressable memories would be useful for four aspects of a system: ready queue, clock wake up queue, I/O device control and resource access control. A preliminary design for CAM hardware is included.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

78 12 18 042

220 200

AB

AD A062462

DDC FILE COPY

DDC RECEIVED DEC 21 1978 F

Hardware Enhancement of Operating Systems

Final Report

Caxton C. Foster

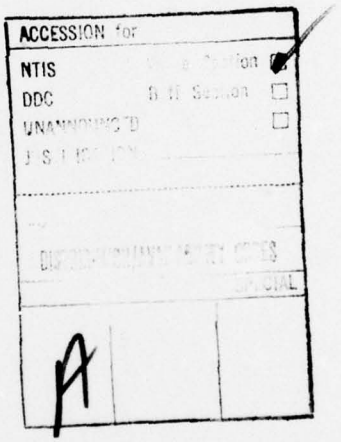
November 23, 1978

U. S. ARMY RESEARCH OFFICE

CONTRACT / GRANT NUMBER
DAAG 29-76-G-0335

UNIVERSITY OF MASSACHUSETTS
Amherst, MA 01003

THE FINDINGS IN THIS REPORT ARE NOT TO BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, UNLESS SO DESIGNATED BY OTHER AUTHORIZED DOCUMENTS.



Introduction

During the past two years we have looked at several ways of improving the performance of an operating system. We have concentrated on improvements that might be brought about by means of additional hardware. Further we have concentrated on small dedicated systems that must respond rapidly to their environment. These are often called real time systems.

Academic types concentrate their academic type research on two aspects of operating systems. These are loosely described as traffic control (semaphores) and as queuing theory. In the real world neither of these has a great deal of bearing on whether an operating system runs rapidly or slowly. To be sure one can eat up a great deal of time by larding a system with complicated interlocks. In the final analysis a great fraction of these semaphores can be eliminated by proper design of the system. One does need to worry about passing messages from one sub program to another and one does need to worry about possible race conditions but both of these can be handled efficiently, tidely and with considerable dispatch by a monitor.

Queueing theory tells one, after a great deal of mathmatical manipulation, that things are going to pile up where bottle necks exist. It can tell you how bad it is likely to get but since one must allow for the worst case in any event this is if only passing interest.

Our conclusions can be stated succinctly. One: supply a good mechanism for holding the queues of processes waiting for resources. Two: provide enough resources so that very few processes are waiting on queues.

Let me come straight to the point. The best hardware enhancement for an operating system that you can buy is more main storage: The reason this is true is very simple. If there is sufficient main storage available you do not need to complicate your system by providing virtual storage. If every-

body is in mainstore all the time, then there is no "unoverlapped swapping time" at all. The control structure is simpler, the operating systems is more reliable and it takes up less of the system's time and resources.

The next best way to spend money on hardware is to buy some content addressable memory to use for storing system queues. This will allow us to post a request for service in one machine cycle and to return the highest priority request-for-service in one cycle. It is hard to imagine how this speed might be improved. It is also hard to imagine how an operating system could get along without having a queueing mechanism of some kind. There are other places to spend money, some of them profitable. One can provide sophisticated memory and I/O device protection hardware. With content addressable memories protection can be provided in a very flexible way that permits each user to have his own access map to storage independent of all other users. Hardware registers to permit renaming blocks of storage, as is done in a virtual memory, turns out to save a lot of moving of information or else waste of space due to external fragmentation. Other than these we found no place that hardware would be significant help to an operating system.

A Small Real Time Operating System Using CAMS

In order to focus our efforts, we decided to concentrate on a dedicated real time system such as might be found in many mini-computer applications.

We assume a system with a limited number of jobs to do. These jobs are known at system generation time. Because this is a real time system, response to external stimuli must be rapid and we must select at any instant the highest priority job ready to run and give that job what it needs to accomplish its purposes. This implies that each job will have a priority that may vary depending on the part of the job being carried out and may vary depending on the imminence of that job's deadline. It further implies that we must be prepared to preempt a job when another one of higher priority becomes unblocked. There will be jobs in the system that may sleep for a long while waiting for an external event but that require high priority service once that event occurs. Other jobs will be awakened periodically so that they may study the state of the world and react appropriately. Still other jobs, such as diagnostic routines, will run only if there is nothing else to do.

Motivation

Brown, et al. (1) identified 15 "system primitives" for an operating system of this type. Furthermore, they picked two, namely queue manipulation and message passing, as especially time critical. Their model showed that if they could reduce the nucleus service time by 70% they could expect a decrease in response time of 25% (2). They were hesitant to claim that the movement of these two primitives to firmware would accomplish this 70% reduction in overall nucleus service time.

An examination of M.M.S. (A Modest Multiprogrammed System, Eckhouse (3)) showed a service time of 240-295 microseconds (on a PDP-11). This system has

only one service call, "queue," which takes the running task, puts it on the ready list and takes the next highest priority ready task and makes it active. This system has no interprocess communication. In addition, MMS has only four levels of priority each on a separate queue so that the Enqueue operations can be done relatively quickly.

Statistics from other operating systems (Foster (4) and Bateson (5)) give a range of "Length of Execution" of system routines (tasks) of 50-200 with a median of around 100 instructions executed between service calls. With an average instruction time for a PDP-11 (on which MMS was written) of 4 μsec, this implies 40% of the CPU time is spent in the operating system nucleus. This agrees well with Brown's 37% estimate.

We believe we can cut down service times for this type of system to 80-100 microseconds for two reasons. First, all Queuing and Dequeuing operations will be done using Content Addressable Memories (CAMS). These will be special purpose hardware that will perform operations on data. Our estimates show that these will interface with the PDP-11 as a 1 microsecond memory and not slow down the execution of instructions.

Second, all other primitives (i.e., block, unblock, "P" and "V" on resources and semaphore, etc.) will also be done using these same CAMS so we expect a similar speed increase for all functions of the system.

An Overview of the Operating System

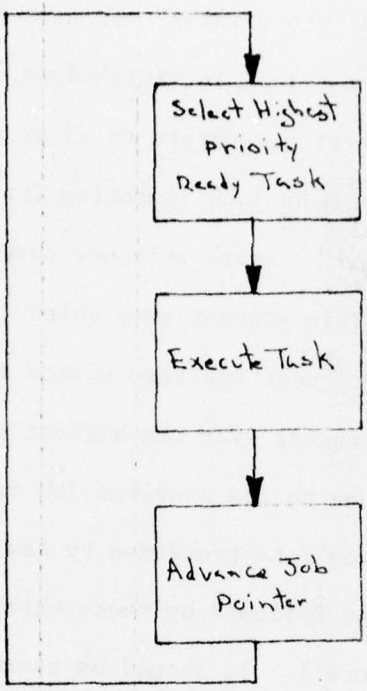
In order to understand the following discussion it is important to understand the concepts of the Job and Task. A task is a small unit of processing; it is typically "request an I/O device," "use an I/O device," or "request a page of memory." Each job consists of a list of tasks to be performed much as a program consists of a list of instructions. Serving as

a pseudo program counter we have a "job pointer" for each job which points to the next task to be executed. When a task is finished we "advance" the job pointer of this job. The job pointer now points to a new task to be performed for this job. But since there are many jobs competing for the use of the CPU we cannot just plunge ahead and start this new task. Instead, we post a "request" on the ready queue. This request says which job wants what task and at what priority. When this request has been posted the system then scans the ready queue and selects the request with the highest priority and executes that. The task selected may belong to the previous job or to another.

Tasks which use resources should be proceeded by tasks which request the assignment of the resource (P) and followed by tasks which release the resource (V). Details may be seen in Figure 1. It should be remembered that all jobs, and tasks, are specified at system generation time. Each task within a job will have its own priority associated with it. There can be several "calls" of a task in the same job (at different points) and in several jobs. We make the stipulation that there will only be one task per job that is active in the system at any one time. By active we mean on the ready list or running. This simplifies the saving of context when a task is interrupted. We allow looping of tasks in a job to accomplish chores such as outputting multiple lines to a line printer.

The ready list or ready queue is kept on one of the four system CAMs. It is a list of tasks with their associated priorities and the number of the job for which they are running. There is also a "status" bit called busy/idle. We make this bit a one if this task is idle. This bit is just above the high order bit of the "priority" field. Thus all idle tasks have a higher "priority" than any busy tasks. This busy/idle bit will be used primarily for tasks that request resources.

(a)



(b)

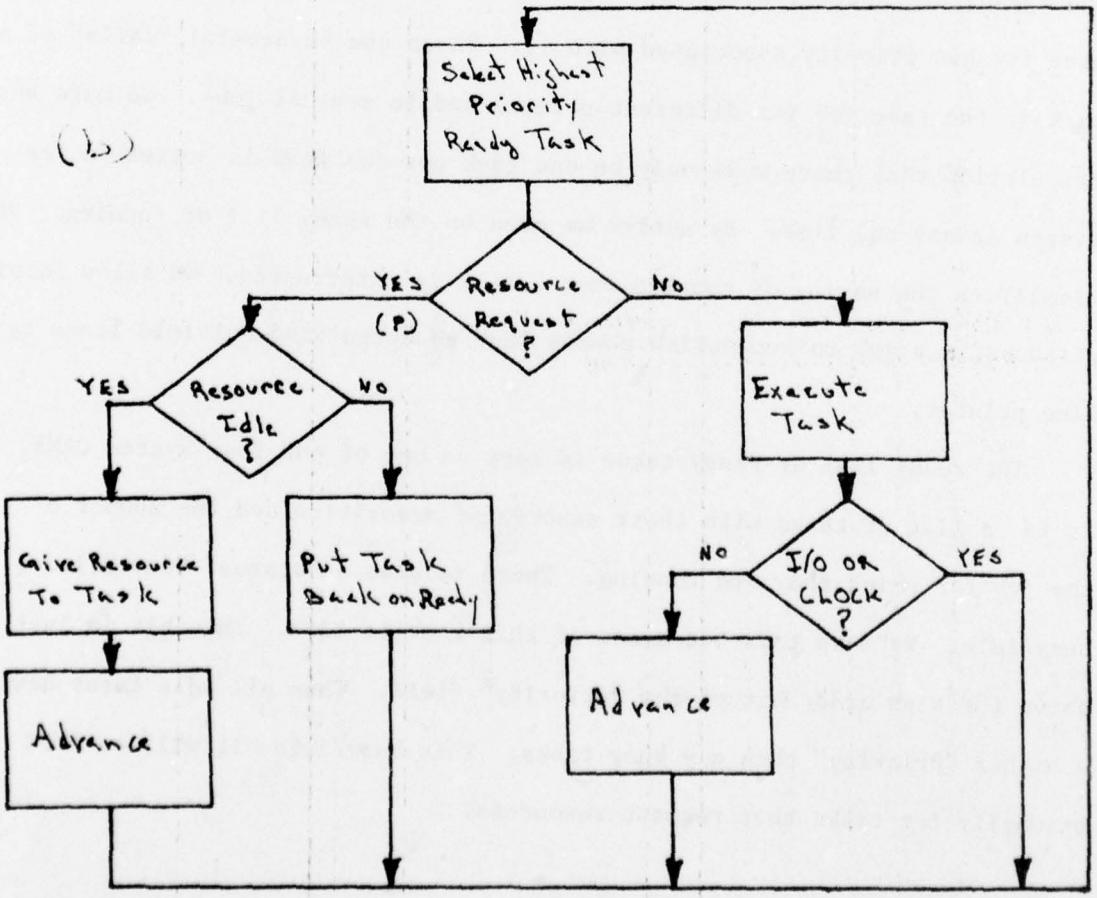


figure 1: (a) Basic execution of operating system
(b) Same in more detail

Main Program

The flow of the "main program" or job scheduler (see Figure 1) is this:

```
do forever
  Begin
  Current-task:= Highest-priority - of (Ready-list);*
  If Task-type (Current-task) = Request - resource
    then begin
    If Is - busy (Resource-Requested by (Current - task))
      then begin
      Task - status (Current - task):= Busy;
      Put - on - ready - list (Current-task);*
      end
      else begin {resource is Idle}
        Disable - interrupts;
        Current - resource:= Resource - requested - by (Current - task);
        for all Tasks where Resource - requested - by (Task) = Current - resource
           do Task - status (Task):= Busy;*
        Update - IO - CAM (Current - task, Current - resource);
        Resource - status (Current - resource):= Busy;
        Enable - interrupts;
        Advance (Current - task);
        end
      else begin {Not a request resource task}
        Execute (Current - task);
          if not (Task - type (Current - task) = Resource - use or
          Task - type (Current - task) = Clock - wake-up)
          then Advance (Current - task);
      end
end
```

*These are "primitive" CAM operations as explained later.

Each resource has two words associated with it: one is the "resource use flag," and one, called "old," is the task and job number of the current user. The execution of a resource use task is:

1. copy the "current " task and job number into "old"
2. set up the I/O conditions (i.e., give a start to the paper tape reader)
3. return to the main loop.

It is assumed that the I/O interface is "smart" enough to generate an interrupt if a sufficient time period has elapsed to justify a time out.

When an interrupt occurs indicating that the I/O is completed, we put the task that was interrupted (called "current") on the ready list as an idle task. Then we do an advance for the "old" task which was waiting for the interrupt. This allows for preemption, since the previously running job (current) and the job which just had an I/O completion (old) from an interrupt can "fight it out" based on their priorities to see which will be taken from the ready list next. For example, a job which just had a task return from a disk interrupt would normally follow that task (on the job list) with a high priority task to use the disk again or release it (with a V task). We would want this to be done before we would want a low priority task (that was running) to finish. We do not want a "valuable" resource like the disk to sit unused with its use flag set busy.

A typical resource release task (or "V") would perform the following:

1. make resource idle
2. make all other tasks on the ready list that are requesting this resource idle (competitors)
3. update the memory protect or I/O protect CAM to show that the resource is no longer owned by this job
4. return to main loop.

Content Addressable Memories (CAM)

Of the four system CAM's two will be used for queuing operations; these are the Ready List and the Clock Wake Up List. Each "queuing CAM" consists of a mico-programmed control register, a data register, a comparand register, and a number of data cells each 24 bits in width. The number of data cells will depend on the application. Each cell is divided into four fields A, B, C, D of 1, 7, 8, 8 bits respectively. See Figure 2.

Each 8 bit instruction to the CAM control word is decoded and the appropriate action is taken on the data cells, comparand, and data register. Instructions to the CAM control word are:

1. Find Greatest

Find the cell with the largest number in the A and B fields (Busy/Idle and Priority) and return the C and D fields of that cell (Task and Job number).

2. Make Tasks Busy

Find all cells that match the C field of the comparand and write a zero in their A field.

3. Make Taks Idle

Find all cells that match the C field of the comparand and write a one in their A field.

4. Clear Entry

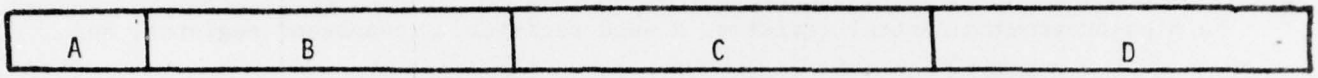
Find the cell that matches the C and D fields of the comparand and write a zero in the D field (there is no job number zero).

5. Make Entry

Write fields A, B, C, D into the (first) cell that has field D set to zero. (If none, this will cause an interrupt.)

6. Exact Match On Time

Find the (first) cell that matches the A, B, C fields of the comparand and return the D field of the cell. (If none return zero.)



Field	For Ready List	For Clock List
A	Busy/Idle	16 bit time
B	Priority	
C	Task #	
D	Job #	Job #

Figure 2

Four Uses for CAM's in an Operating System

We have found four places within a conventional operating system that would profit from the use of CAM's such as we described above. There may be more but these four are obvious. If once a machine was built with some CAM's in it, we believe that more ways to use them would be discovered very shortly. The four places we have investigated are:

job-task request list

clock wake up list

memory access control

I/O device access control

We will now discuss these applications in turn and explain some of the reasoning behind our choices of parameters.

Job-Task Request List

As outlined above, each job has a task it wishes to perform. The collection of these task requests is called the "ready queue." This task might be "waking up" at a certain time or waiting for an external event or computing a square root. Associated with each task entry on a JOBLIST, we have a word describing the priority at which that task needs to be run for this job. Because the PDP-11 has 8 bit Byte instructions, we have elected to make our CAMs a multiple 8 bits, namely 24 bits, wide. We have allowed:

8 bits for 256 different jobs

8 bits for 256 different tasks

7 bits for 128 different priority levels

1 bit for Busy or Idle

In a straight conventional computer, it would be necessary to either keep the "ready queue" sorted by priority, search the queue for the highest priority task each time we wish to dispatch a task or have 128 separate queues.

An additional complication for a conventional machine is the status (busy or idle) of a task. The usual way to handle this is to provide a "blocked" queue for task requests that are waiting for "busy devices." When a device falls idle a conventional operating system must search the blocked queue for the highest priority request waiting for this device and transfer that request to the ready queue where it then competes with all other ready requests for the CPU. If an interrupt occurs which triggers off a high priority request for a device, caution must be exercised to ensure that not more than one request for a possibly busy device appears on the ready list.

The use of a CAM in this place greatly simplifies the situation. When a request for the execution of task is generated (by the execution of the ADVANCE routine), we insert a task request in the CAM with the job number, task number and priority as determined by ADVANCE. We set the status bit of the request to 0 or 1 depending on whether the task is known to be idle or busy. Since both the blocked and the ready queues are in one CAM, there will be no need to move entries between them.

When we are ready to dispatch a new task, we search (in about one microsecond) the entire CAM for the request for an idle task which has the highest priority level. This task is the one to be dispatched and the request for this task is removed from the CAM. If this task involves a possibly busy device (for example, a line printer) when the device goes busy part of the code for the task will access the CAM and set all requests for this task busy. When the device falls idle, we simply perform the inverse operation marking all requests for this task as idle and hence as candidates for dispatching.

When an interrupt occurs, we post a request in the CAM for the task that was currently executing and make it an idle task. Then when the interrupt handler is finished, we go back to the dispatcher and if the task which got

interrupted has the highest priority, we go back to that task. Otherwise, we go to some other task with a higher priority.

Now suppose we are executing a task for a job, J, involving the line printer and an interrupt comes along saying that the disk has finished a transfer for job K. Job K now needs to use the line printer. It look in the printer status word and finds the status is busy so it enters its request in the CAM as a busy request. When J was interrupted, it stored an "idle" request in the CAM so when the interrupt has been serviced, we guarantee that J will be the first one to get the line printer, line by line interlace of two jobs not being the most propitious way to employ a line printer.

The reader will note that the use of a CAM has not reduced operating system design to child's play but that the dispatching and interrupt routines have been considerably simplified by its use.

Clock Wake Up List

The second place we have found CAMs useful is in the clock wake up list. Many tasks need to be activated periodically or after a certain amount of time has elapsed. Meanwhile they must sleep quietly using as few resources as can be arranged. Typically, they sleep on a clock-wake-up list. When the real-time clock interrupts, this list is examined to see if any job needs to be awakened at this time. Once again, either the list must be kept ordered or it must be searched. The use of a CAM elimiates both requirements. One task available in our system will be of the form "wake me up after N time units have elapsed," where N is an integer less than 65,536. Current time is kept as a 16 bit number. This number is added to N and the sum modulo 2^{16} is stored in fields A, B, and C of the request. The number of the job making the request goes in field D.

On each tick of the clock, we interrogate the CAM to see if any requests

for this time are posted. If there are, we do an ADVANCE for each job desiring to be woken up at this time. With a 60 cycle clock 64K time periods come to 18.2 minutes. Sleeping periods longer than this must be created by using two or more maximum periods. Provisions for looping in a job list have been included in the system design.

Since the clock interrupt will occur with high frequency, the simplicity of this scheme will have relatively large payoffs in system efficiency.

Memory Access Control

In a dedicated system such as the one we are concerned with, it would seem reasonable to store most programs in ROM and keep them permanently resident. Nonetheless, working space can sometimes be profitably shared between two or more jobs and a provision for controlling read-write storage access can serve as a prototype for the handling of other resources of similar character.

Each job in the system is assigned a job number from 1 to 255. Job number zero is reserved for the system itself. We will add a "peripheral device" to the PDP-11 which is actually a register to hold the current job number.

We will divide main memory into pages of 128 words (256 bytes) each. With an 18 bit main memory address this allows a potential 1024 pages.

When a memory reference occurs, the 8 bit current job number and the 10 bit number of the page addressed are concatenated into an 18 bit word and presented to the memory access control CAM. If there is an entry in the CAM which corresponds to this 18 bit comparand, this means that this job is permitted to access this page and the reference is permitted to proceed. If there is no such entry then the memory reference is inhibited and an interrupt is generated. When job number zero is detected, the CAM is bypassed and the

system is given permission to access any memory cell. We will make the CAM large enough so that entries can reside there semipermanently and in fact so that more than one job can have access to a page in case we wish to share a page between two or more jobs. If most programs are stored in ROM, we do not need to protect them from being overwritten and the CAM may be made correspondingly smaller to give protection only to read/write memory.

I/O Control

In the PDP-11, all peripheral devices are given memory addresses in the range from 760000 to 777777 providing 4096 possible addresses or 8192 bytes. To say this another way, when bits A<17-13> are all ones then bits A<12-1> select one of 4096 device registers not all of which actually exist in any given installation. We will break these 4096 registers into 1024 blocks of four registers each and provide protection for each such block present in the system. Each I/O device is assigned to a block or blocks and its registers are all in two or more adjacent blocks. Examination of DEC's peripheral address assignments indicates that this will be in accord with their recommendations. With this scheme, we can assign I/O devices to specific jobs without interference. The 8 bit current job number is concatenated with the ten address bits A<12-3> and presented to an I/O control CAM. The action of this CAM is then identical to the memory access control CAM just described. We must select which of these two CAM's is to control access on the basis of the high order address bits A<17-13>. If these are all ones, then the I/O control CAM is used. If not, the memory access control CAM is brought into play. Figure 3 shows in schematic form how this might be accomplished.

Conclusions

We have examined four places in which content addressable memories could

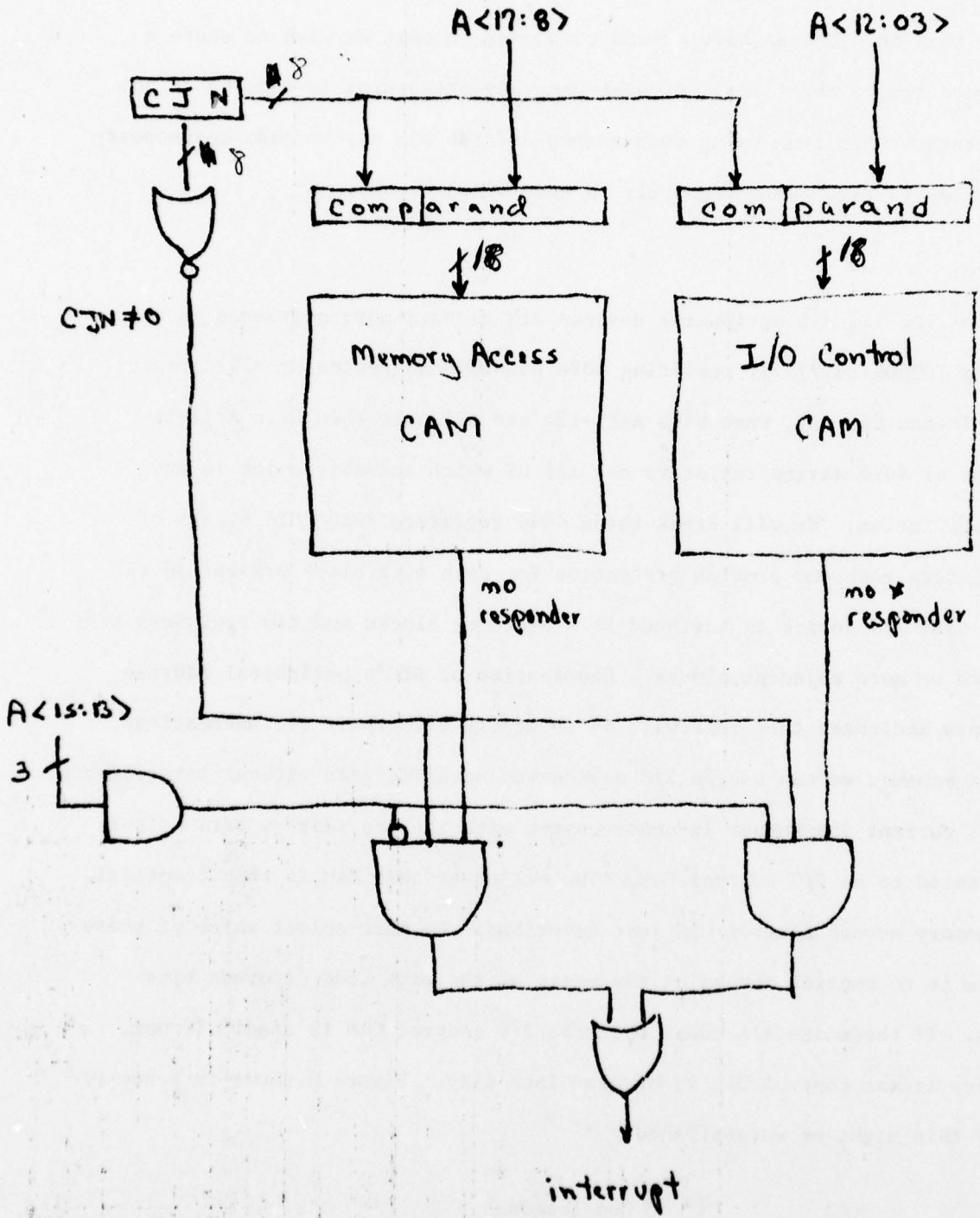


Figure 3. Schematic diagram of how the Memory access and I/O control CAMS generate an interrupt if permission to use a page or an I/O device does not exist.

substantially enhance the behavior of an operating system. Used to implement the ready and blocked queues and the clock wake-up list, they offer increased speed and simplified software. Used for access control of memory and I/O devices, they provide services not presently implementable on a PDP-11 in real time. There are undoubtedly other places where CAMs could be used and as we proceed to write the operating system described above, we may discover them.

References

1. Brown, George E., Eckhouse, Richard H. (Jr.), Goldberg, Robert, P. (1975) Operating system enhancement through micorprogramming. CENTACS Report No. 52, U.S. Army electronics Command, Fort Monmouth, New Jersey. August.

2. Brown, George E., Eckhouse, Richard H. (Jr.), Estabrook, Jay A. (1976) Operating system enhancement through microprogramming: design and implementation. CENTACS Report No. 53A, U.S. Army Electronics Command,.Fort Monmouth, New Jersey. November.

3. Eckhouse, Richard H. (Jr.) (1975) Minicomputer Systems: Organization and Programming (PDP-11). Prentice-Hall, Inc.: Englewood Cliffs, New Jersey. Chapter 9.

4. Foster, Caxton C. Private communication.

5. Bateson, A.P., Ju, S.M., Wood, D. (1970) Measurements of segment size. Comm. ACM 13, 3. March. 155-159.

Preliminary Design of CAM Hardware

In this section we will discuss the design of the Content Addressable Memories (CAM). The design goal that we used were:

1. Minimal Chip Count
2. Reasonable Fast Speeds
3. No Custom parts (Standard TTL Components)

Obviously goals 1 and 3 conflict with number 2. As a compromise we decided to implement the CAM with a bit serial, word parallel architecture. This means that each word rather than each bit has "local intellegence." This minimizes the amount of comparison circuitry needed. It also means that some speed is sacrificed in that each CAM operation is done on each bit of each word in a serial fashion even though the operations are done on all words in parallel.

The cycle time (per bit) was chosen as 100 nano seconds. This is within the speeds of Schottky TTL and gives adequate overall performance. It was also decided to micro-program the CAM operations rather than use discrete logic. The micro-program controller will be able to operate at 10 MHZ speeds and it will allow maximum flexability for future changes to the system. It will also allow for CAM operations other than the ones needed specifically for the Operating System to be developed as needed with little or no changes to the basic hardware.

The CAM is designed in 64 word x 32 bit blocks. 32 bit words are used because we know we need at least 24 bits for the Operating System operations and 32 is the next multiple of 16 (the width of the words in the PDP-11). The 64 word block is not as limiting as it sounds, the blocks will be able to be cascaded to any length up to 16 (1024 words). The size of 64 words is convenient size to put on one printed circuit or wire wrap card. (about 120

packages). We propose to build 12 of these cards plus the one controller card that will drive all of them (independently or in groups) and interface to the PDP-11 Unibus.

The CAM word is designed around a dual 32 x 1 bit Schottky TTL RAM chip (Signetics 82521, 86521). This memory was chosen because of it's ability to write independently into either (or both) of it's two 32 x 1 memories. These have typical access times of 25 nanoseconds (50 max). As can be seen in figure 1, each word also needs an Exclusive-Or gate to compare (bit serially) the contents of the CAM word to the contents of the chosen comparand register. Additionally each word needs to have a gate to conditionally allow its contents to be read out of the CAM. Note that we have implemented a multi-read as well as multi-write capability. That is, all words that have their tag bits set will participate in read and write operations.

The tag register is built with 74279 Quad $\overline{S}$-$\overline{R}$ latches. S-R latches are used since any single bit mis-match between the comparand and the CAM word should reset the corresponding tag register bit (Tn) for that word. The S inputs are used to accomplish the "set all" Command. Figure 2 shows the tag register and the select first (Sel 1st) logic which is used to pick the first responder and reset all the tag bits of "later" CAM words. 745158 Quad 2 input multiplexer circuits are used to select between the mismatch signal and the select first chain to generate the resets to the Tag bits. A strobe signal is used to enable these reset signals only after a delay (about 50 nsec. for mismatch and about 500 nsec. for select first) to allow for gate propagation times.

Although the select first chain could give us the answer to the question "are there any responders" the propagation time is too long for the cycle time of the CAM so there is another set of logic (figure 3) to monitor the tag bits and generate a SOME/$\overline{\text{NONE}}$ signal. This logic tree will be able to

respond fast enough to be used as a test by the micro-controller. Not shown in the diagrams is the logic to connect several (up to eight) of the 64 word blocks together to allow for a maximum of 1024 words. Also there will be circuitry to cascade the select first and SOME/$\overline{\text{NONE}}$ circuitry.

The control circuitry is shown in figure 4, 5 and 6. Figure 4 indicates the interface to the PDP-11 Unibus, the comparand registers, the comparand multiplexers, the read decoder, and the CAM control word. The micro code can select one of four 8 bit comparands to be compared against the contents of the memory in either true or negated form (CØ is simply a source of zeros). The read decoder is necessary since the data is read out of the CAM words in a bit serial mode. The data in the CAM control is used as a branch address in the micro code. This is similar to the Branch On Op-Code (BOOC) operation used in microprogramed computers. Not shown is the additional circuitry necessary to specify which subset of the up to 16 64 word block are to be operated on. This will be done with discrete logic and will be transparent to the micro controller.

Figure 5 shows the micro controller itself. The microprogram will be stored in 74S471 type PROM or 74S371 type ROM chips. A preliminary count shows that about 128 instructions will be needed to execute the six CAM operations for the Operating System. We have allowed for 256 instructions so that we may add other operations as needed. Typical branch control circuitry is shown in figure 6 and the format of the microprogram word is shown in figure 7. There is extra room in the micro word to add other controls and tests as needed for other CAM operations. The following are outlines f[illegible]ecution of the six known CAM operations.

1. Find Greatest (used to find the highest priority Task in the Ready List CAM)

Typical PDP-11 Calling sequence:

```
MOVB  #FGTST, CCW
MOVB CDATA, TASKN
MOVB CDATA, JOBN
```

CAM Algorithm*:

```
SET TAG
WRITE (CØ, 31)          ; Bit 31 is used as a flag
COMPARE (C̄Ø̄, Ø)         ; Busy/Idle Bit
WRITE (C̄Ø̄, 31)          ; Keep only those that are Idle
FOR I = 1 to 7 DO       ; * *
BEGIN
     SET TAG
     COMPARE (C̄Ø̄, 31)     ; See if any active cell
     COMPARE (C̄Ø̄, I)      ; has this bit set
     IF "SOME" THEN       ; yes
     BEGIN                ; make others inactive
          SET TAG
          COMPARE (C̄Ø̄, 31)
          COMPARE (CØ, I)
          WRITE (CØ, 31)
     END
END
SELECT 1st                ; To read out the RASKN and JOBN
ZERO CDATA                ; It is an RS latch
LOOP ON SELECT 1st TIMER  ; for propigation delay
FOR I = 8, 15 DO
     READ (I)             ; TASKW
```

```
ZERO CDATA          ; ***
FOR I = 16, 23 DO
     READ (I)       ; JOBN
SET TAG
WRITE (CØ, 31)      ; Reset all Flags
BRANCH TO NOP       ; for new CAM operation
```

* For the following Algorithms the following conventions apply. WRITE (CX, Y) means to write into CAM Bit address Y the corresponding bit from comparand CX. $\overline{CX}$ implies inversion. A similar meaning is given to COMPARE (CX, Y). READ (Y) simply specifies the bit address of the CAM words that will be read.

** For simplicity "For" loops are shown. In actuality these loops will be unwraped in the micro-code to speed up operations.

*** Handshaking with the PDP-11 is not shown but it will be done in the micro-code by means of the MASTER SYNCH test and the SLAVE SYNCH CONTROL BIT.

1A. Find Greatest (this version is faster, but it assumes that the priority is coded as one bit per priority level rather then levels 0-128). Since the 11 allows only 4 levels of priority there is some justification to assuming that 7 levels will be enough for most applications.

```
FOR I = 1, 7 DO
BEGIN
     SET TAG
     COMPARE (C̄Ø̄, 0)
     COMPARE (C̄Ø̄, I)
     IF "SOME" THEN
          BRANCH TO L1
END
L1:  SELECT 1st
     (* READ IS THE SAME *)
```

2. Make Entry (used to post a request in the Ready List CAM, or Clock Wake Up CAM)

```
PDP:  MOVB  B/I - Priority, C1*
      MOVB  TASKN, C2*
      MOVB  JOBN, C3
      MOVB  #ME, CCW
Algorithm:
      SET TAG
      FOR I = 16, 23 DO
           COMPARE (CØ, I); Find Job Ø the Null Job
      SELECT 1st
      IF "NONE" THEN
      BEGIN
           INTERRUPT; *
           BRANCH TO NOP
      END
      LOOP ON SELECT 1st TIMER
      FOR I = 0, 7 DO
           WRITE (C1, I)
      FOR I = 8, 15 DO
           WRITE (C2, I)
      FOR I = 16, 23 DO
           WRITE (C3, I)
      BRANCH TO NOP
```

* C1, C2 will get the upper and lower byte of the Time for the Clock Wake Up List.

** If the Ready List CAM is full (no Null Jobs) the micro-controller will inform the Operating System by generating an interrupt.

. Clear Entry (used to remove a request from the ready list or the clock wake up list CAM)

```
PDP:  MOVB TASKN, C1
      MOVB JOBN, C2
      MOVB #CE, CCW
Algorithm:
      SET TAG
      FOR I = 8, 15 DO
           COMPARE (C1, I)
      FOR I = 16, 24 DO
           COMPARE (C2, I)
      FOR I = 16, 24 DO
           WRITE (CØ, I); NULL JOB
      BRANCH TO NOP
```

4. Make task idle

```
PDP:  MOVB TASKN, C1
      MOVB #MTI, CCW
Algorithm:
      SET TAG
      FOR I = 8, 15 DO
           COMPARE (C1, I)
      WRITE (CØ, 0)
      BRANCH TO NOP
```

5. Make task busy

```
PDP:  MOVB TASKN, C1
      MOVB #MTB, CCW
Algorithm:
```

```
        SET TAG
        FOR I = 8, 13 DO
             COMPARE (C1, I)
        WRITE (C̄Ø̄, 0)
        BRANCH TO NOP
```

6. Exact match on time (for the clock wake up list)

```
PDP:    MOVB TIMEØ, C1
        MOVB TIME1, C2
        MOVB #EMT, CCW
        MOVB CDATA, JOBN
```

Algorithm:

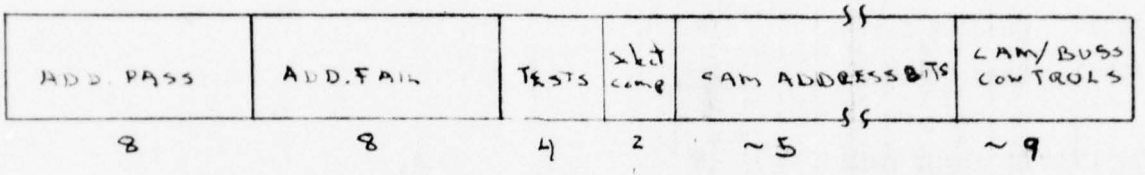
```
        SET TAG
        FOR I = 0, 7 DO
             COMPARE (C1, I)
        FOR I = 8, 15 DO
             COMPARE (C2, I)
        SELECT 1st
        ZERO CDATA
        IF "NONE" THEN
             BRANCH TO NOP
        LOOP ON SELECT 1st TIMER
        FOR I = 16, 23 DO
             READ (I)
        BRANCH TO NOP
```

Package Count 64 x 32 CAM

82521	Dual 32 x 1	32
74586	Quad XDR	16
74503	Quad Wand	16
		64
745158	Quad 2:1 max	16
74279	Quad $\overline{S}\overline{R}$ latch	16
74532	Quad or	16
		48
745133	13 input nand	5
74504	Hex inventor	1
74530	8 input nand	1
		7
		119 packages

M - Program Word of CAM CONTROLER

~40 BITS X 256 WORDS



NOTE: LOW ORDER
3 BITS OF
CAM ADDRESS
Go TO Broadcast
Compared Mux and
Read Data Decoder.

TESTS: Some/word Line
Select 1st Timer
Branch On CAM Control Word
Master Synch From PDP-11

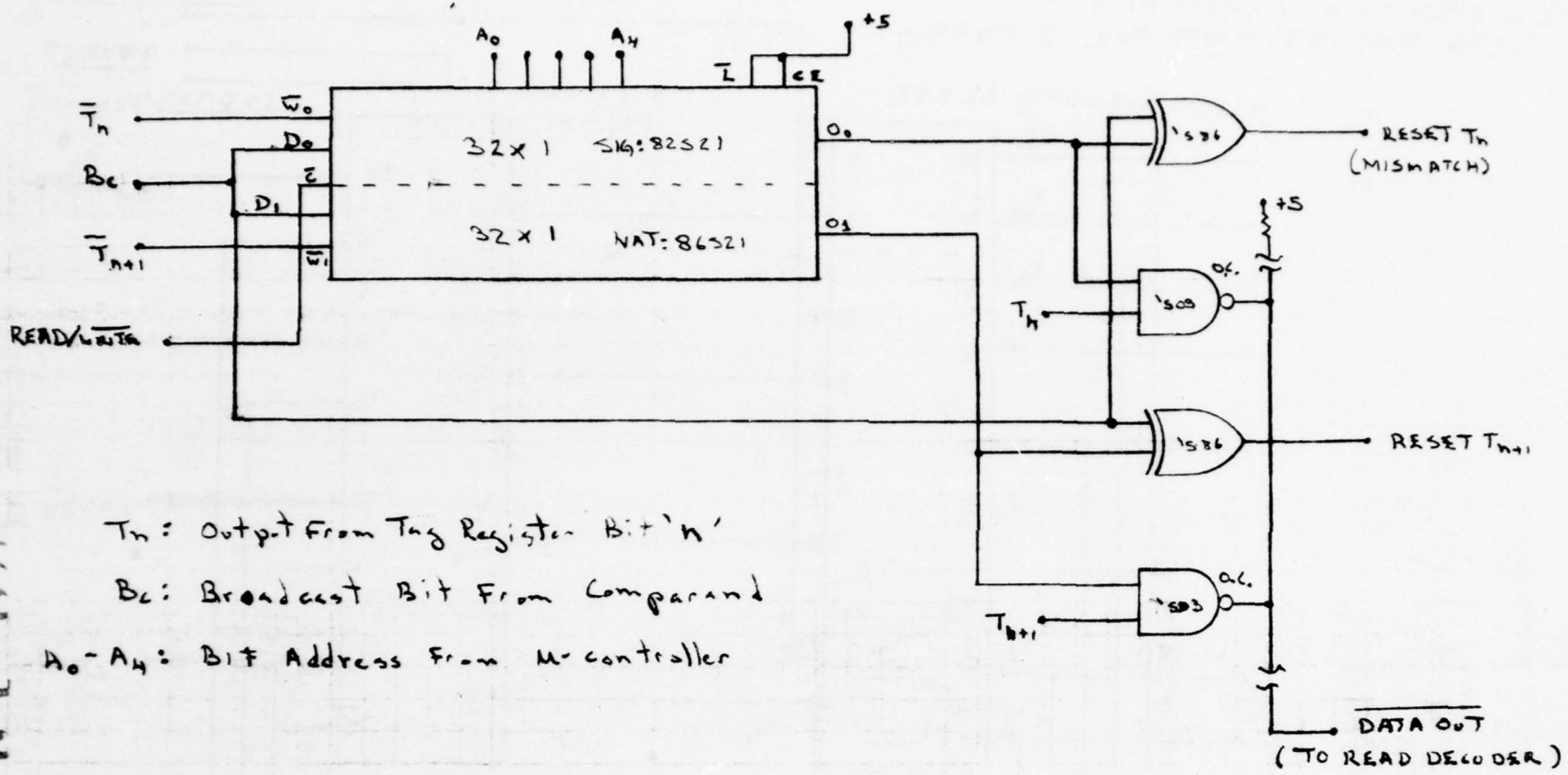
CAM CONTROLS: Selct 1st Responder (Enable Tag Clock)

Set TAG REGISTER
ZERO (-) DATA REGISTER
COMPARE (Enable Tag Clock)
Broadcast compared Inverse
READ

WRITE

PDP-11 BUSS
CONTROLS : SLAVE-SYNCH

Interrupt



T_n: Output From Tag Register Bit 'n'

B_c: Broadcast Bit From Comparand

$A_0 - A_4$: Bit Address From Microcontroller

FIGURE 1 2 WORDS OF CAM MEMORY PLUS COMPARISON, READ AND WRITE LOGIC

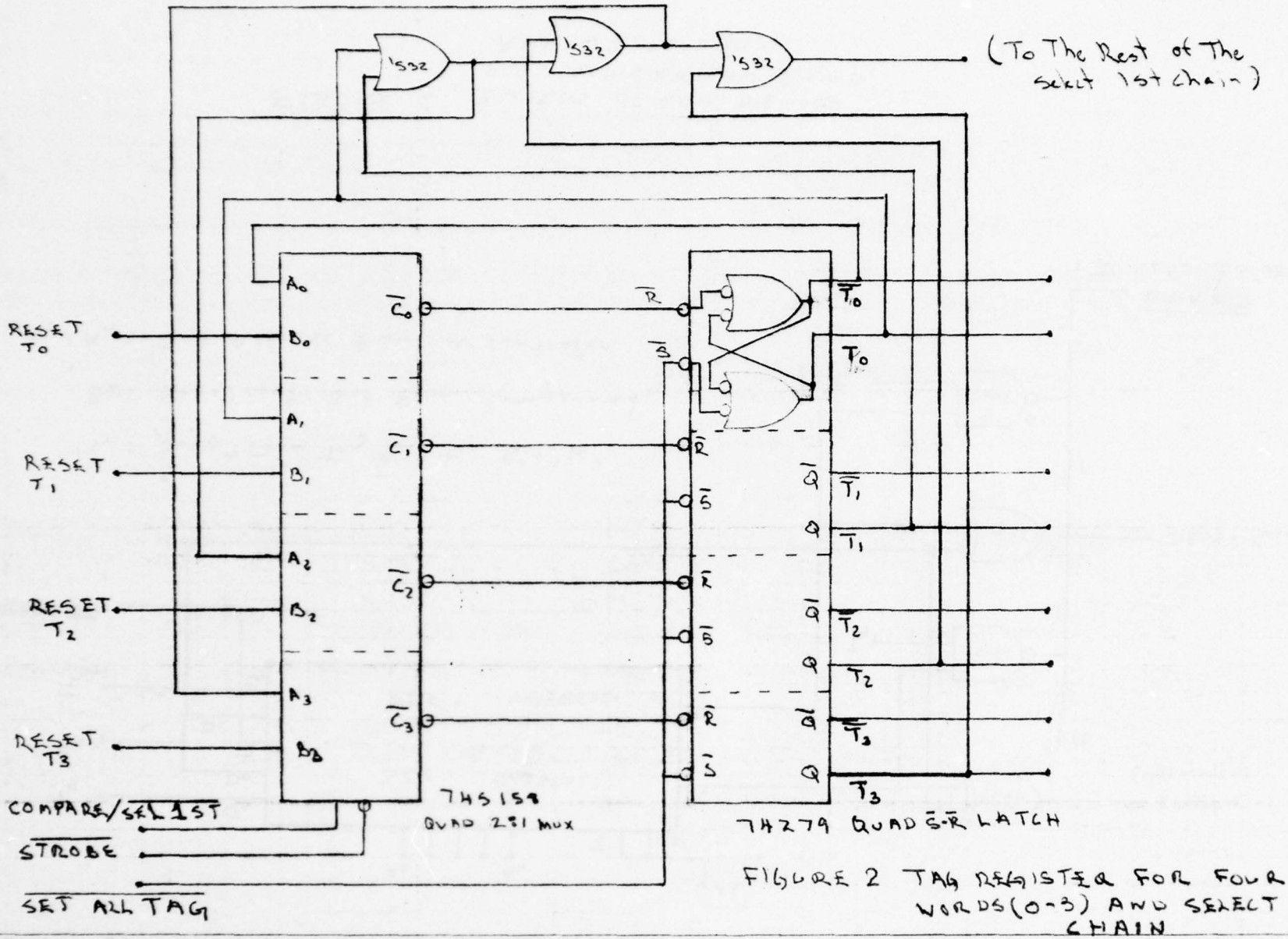


FIGURE 2 TAG REGISTER FOR FOUR WORDS (0-3) AND SELECT 1ST CHAIN

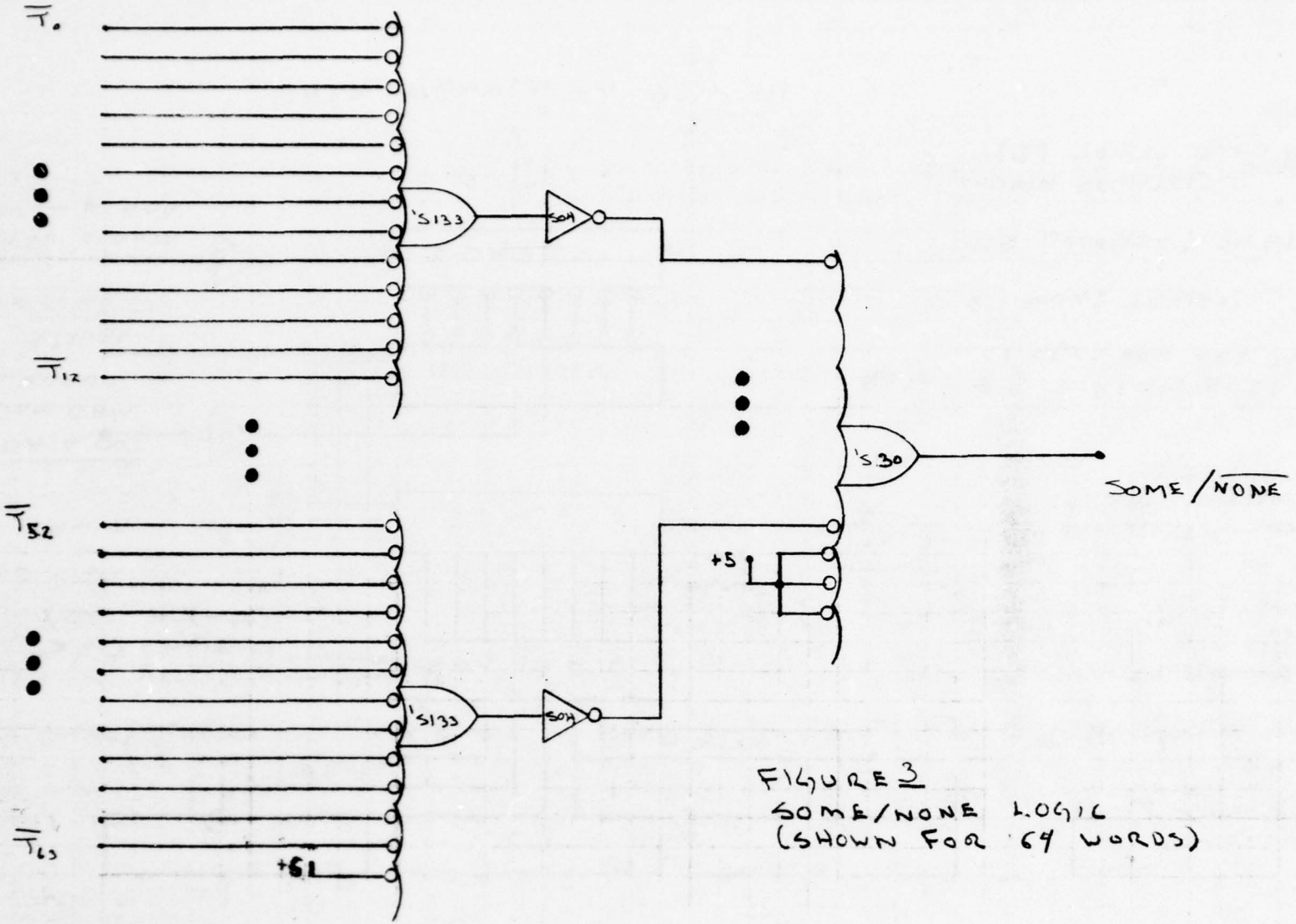
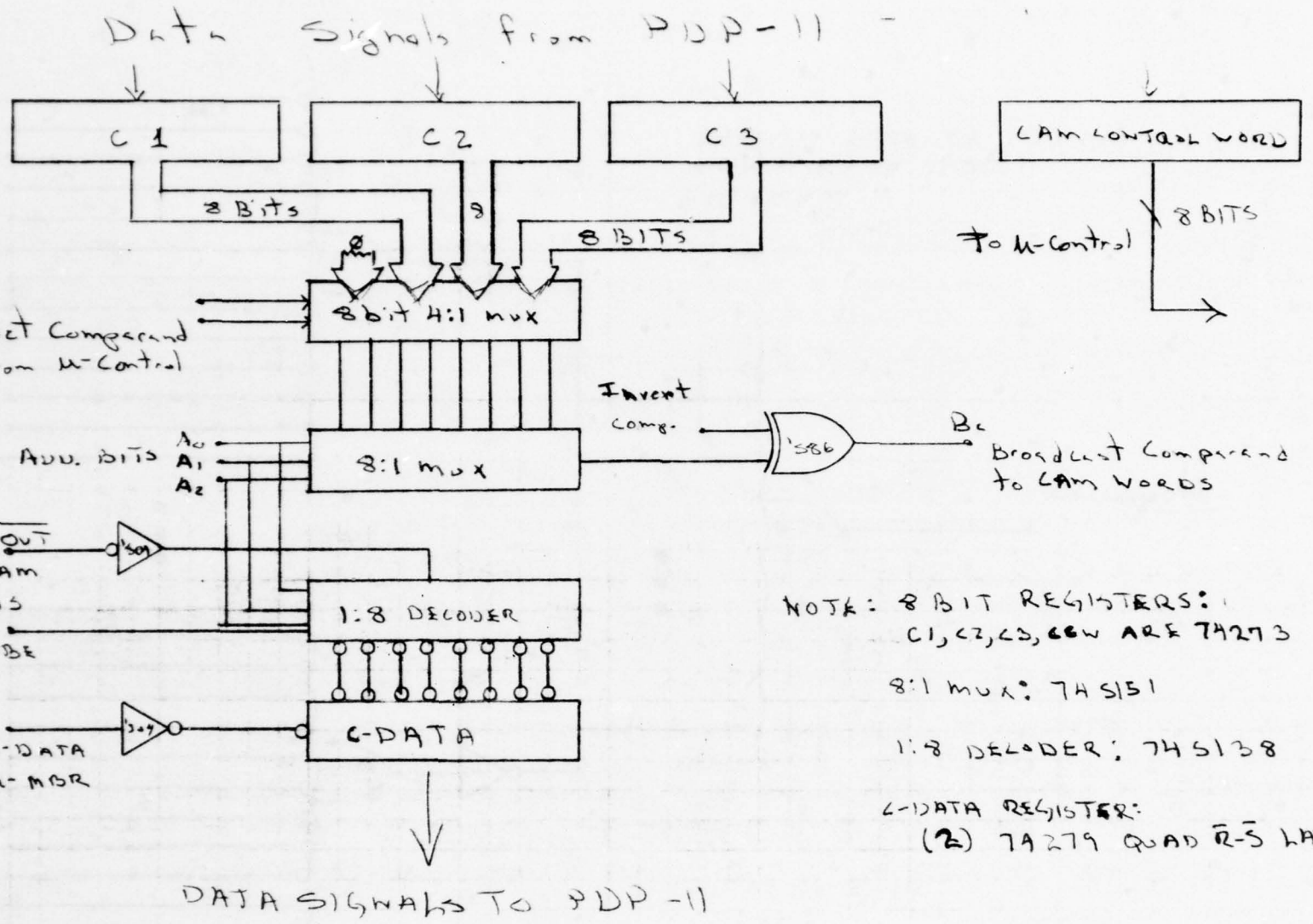


FIGURE 3
SOME/NONE LOGIC
(SHOWN FOR 64 WORDS)



NOTE: 8 BIT REGISTERS:
C1, C2, C3, CCW ARE 74273

8:1 MUX: 74S151

1:8 DECODER: 74S138

C-DATA REGISTER:
(2) 74279 QUAD $\bar{R}$-$\bar{S}$ LATCH

FIGURE 4. COMPARAND, CAM CONTROL WORD AND READ LOGIC

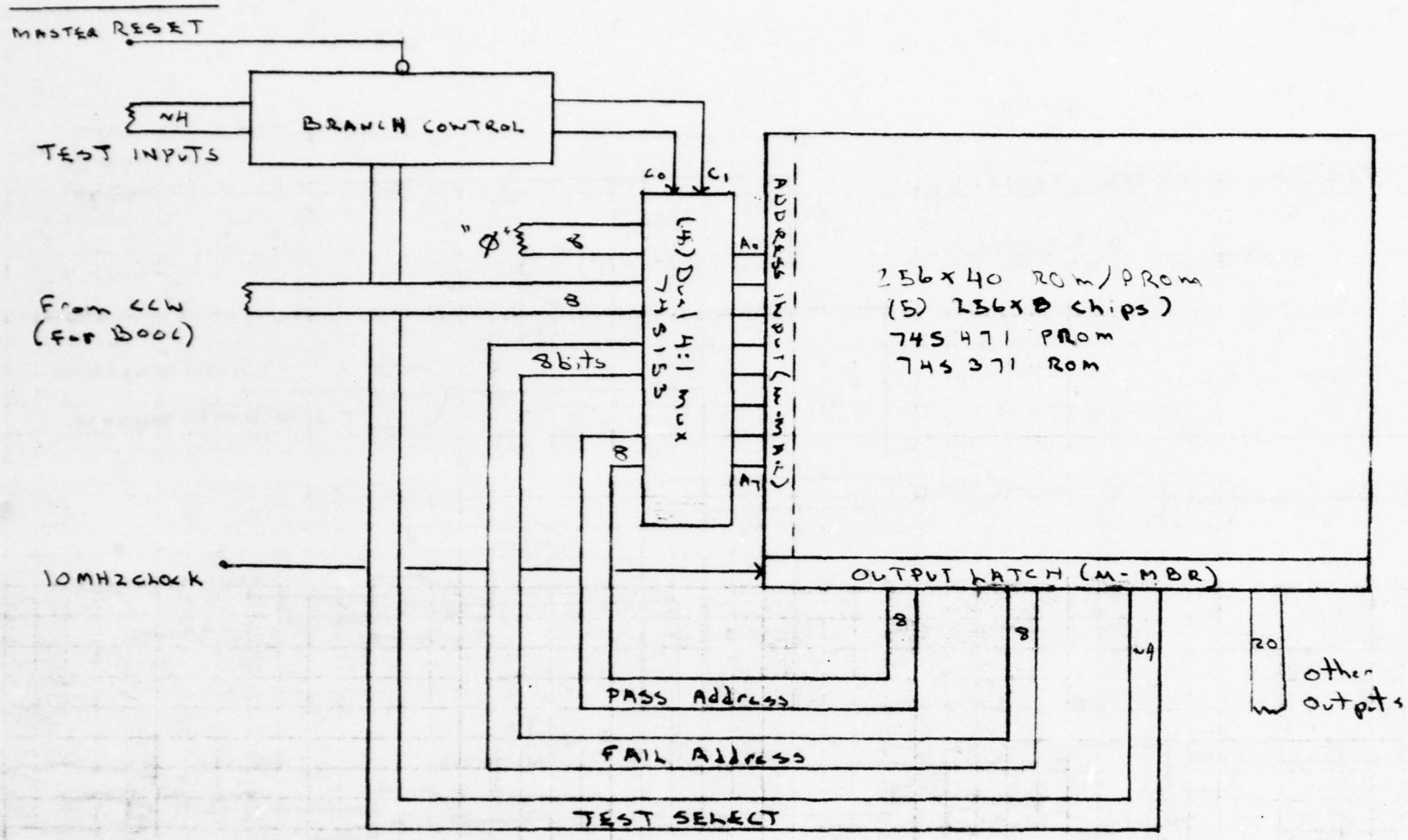


FIGURE 5. MICRO-PROGRAM CONTROLLER

NOTE: OUTPUT LATCH:
(5) 74S273 OCTAL D-FLIP FLOPS

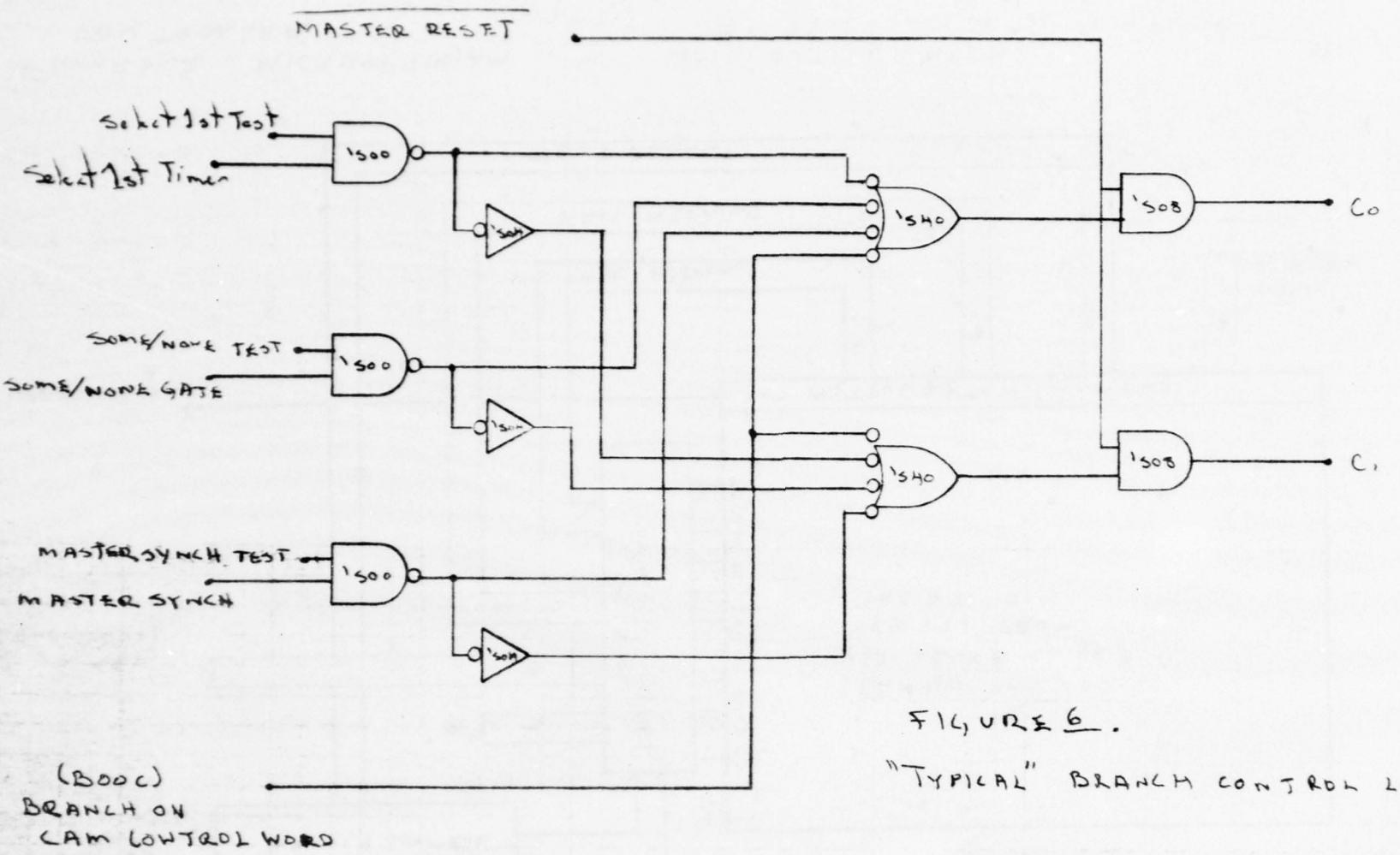
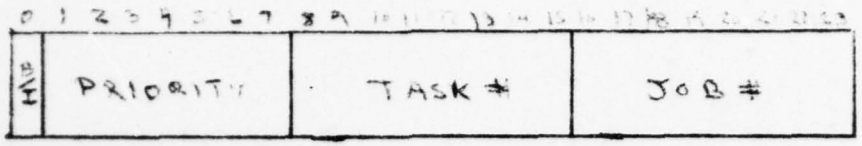


FIGURE 6.

"TYPICAL" BRANCH CONTROL LOGIC



1. FIND GREATEST: 8 bit Maximum bits 0-7; *; Read: 8-15, 16-23
   1600 nsec 800 nsec 800 nsec = 3.0 μsec
2. MAKE ENTRY: 8 bit search=0; *; WRITE: 0-7, 8-15, 16-24
   800 nsec 800 nsec 800 nsec. 800 nsec = 4.2 μsec
3. CLEAR ENTRY: 16 bit search=CCR; WRITE: 0→ 16-23
   1600 nsec 800 nsec = 2.4 μsec
4. MAKE TASK IDLE: 8 bit search=CCR; WRITE: 1→0
   800 nsec 100 nsec = .9 μsec
5. MAKE TASK [illegible]: 8 bit search=CCR; WRITE 0 [illegible]
   800 nsec 100 nsec = .9 μsec
6. EXACT MATCH TIME: 16 bit search=CCR; *; Read: 16-23
   1600 800 = 3.4 μsec

*: select 1st ≈ 1000 nsec

Power Requirements (Typ.)

	Each	xN	
82521			130 MA
74586	250	8000	MW
74503	70	2240	
745157	250	8000	
745175	300	9600	
74532	140	4480	
745133	19	209	
74504	114	228	
		32,757	MW

32.757 Watts

I = 6.55 Amps at 5 volts

Brief summary of the Kronos operating system

1) Overview:

KRONOS 2.1 is one of the operating systems for the CYBER computer family. It accepts jobs from batch, remote batch and time sharing. Its design goal seems to be a compromise between giving a reasonnable response time to interactive jobs and insuring an overall high throughput.

The system overhead in the central processor is limited in comparison to other operating systems since a large number of system funetions including scheduling, are performed by peripheral processors. As a result, KRONOS has a rather high degree of multiprogramming.

2)System Request mechanism:

When a program makes a request, it posts a description of the request in a fixed location of its address space (i.e. RA+1). The program can either initiate a context swap (XJ instruction) to the system or continue processing until the request has been taken into consideration by the peripheral processor monitor. For most requests, the program is suspended from the time it initiates the context swap (or from the time the request is taken into consideration by the system) until the time the request is satisfied. This feature is called auto-recall. For data transfer requests, the program can continue processing while the request is performed. It is possible for the program to check from time to time whether the request has completed. These requests are said to be honored without auto-recall.

The context swap issued by a program exchanges the CP registers with a block of 16 words in the system protected address space. This feature allows the Central Processor monitor to take the request into account much faster than PPU Monitor. If the request is without auto-recall and the program issued a context swap, the control is returned to the program once the request has been initiated.

3) Basic program environment

A job is a sequence of program steps as indicated by a sequence of control cards (if the job is from batch or remote batch origin) or by a sequence of time sharing commands. The only way to preserve information from one program step to the next is through temporary or permanent program files.

3.1. Memory management:

When a program is loaded, it is allocated a fixed, contiguous part of the central memory, called Field Length (FL). The program address space maps directly into this portion of the central memory. The RA internal register contains the absolute starting address of the field length and is added to every address expressed during program execution. The FL internal register contains the maximum displacement allowed in the program address space and prevents the program from addressing outside of its assigned bounds. The only memory request that a program may issue is MEMORY. Depending on the parameters, MEMORY will extend or reduce the program field length at its high end. Any other system action like compaction of the central memory or swapping of the active program is totally transparent to the program.

3.2 File Management and Data transfers:

KRONOS provides an extensive repertoire of file management requests. Since these functions are similar to the ones in other systems, they are not reviewed here.

Data transfer requests are simple record oriented transfers (WRITE and READ, with a few variations) between mass storage and central memory buffers. When issuing a data transfer request, a program has the option to suspend itself (request with auto-recall) or to continue computing and periodically check for completion of the transfer (request without auto recall). In the later case, the program can suspend itself until completion of the request (by issung a RECALL request), once no further processing is possible.

3.3 Task Management:

There is no provision for quasi parallelism within a program in

KRONOS. From the system point of view, each program is a single task. However, KRONOS provides facilities for a user to decompose his program into an"executive task" and various subtasks. The "executive" has complete control of the "subtasks": within the time slice allocated to the program, the user executive can allocate sub-time slices to the subtasks; the user executive can protect itself from the subtasks and can protect the subtasks from each other; the user executive intercepts all system requests issued by the subtasks.

Besides these pseudo tasking facilities, KRONOS enables a program to suspend itself until a previous request is completed or for a certain duration of time (RECALL requets). The ROLLOUT request is similar to RECALL in that the program is suspended until a previous request can be satisfied, an external event occurs, or until a certain amount of time has elapsed. ROLLOUT differs from RECALL in that the program is swapped from the central memory. From the system point of view, ROLLOUT is preferable to RECALL when the expected waiting time is long. From the program point, ROLLOUT and RECALL are logically equivalent.

Another feature relevant to task management is the possibility for a program to create another job through the SUBMIT request.

3.4 Error handling:

The MODE request allow a program to specify which central processor exceptions (e.g. division by zero, indefinite operand, address out of bounds) should cause termination of the program and which exceptions are to be completely ignored by the system.

The ERREXIT request facilitates a limited amount of error recovery processing. When issuing the ERREXIT request, a program specifies a single entry point to which control should be passed when a fatal error occurs. Fatal errors include not only fatal central processor exceptions but also exceptions that occur during processing of other requests like illegal PP or CP requests, time limit, exceeded maximum number of files, or exceeded file space limitations.

4) Subsystem Environment:

KRONOS provides additional facilities for programs that are extensions of the operating system itself, like TELEX, the time-sharing executive, or TRANEX, the transaction executive. These facilities can be roughly classified between extensions of the memory management, file management and interprocess communication primitives.

4.1. Differences in memory management:

Subsystems are programs explicitly known by the system and are given the highest priorities. They are permanently resident in central memory and cannot be swapped.

4.2. Differences in File management:

Subsystems have access to priviledge files and to the files that have been submitted to the system for execution or that result from the execution of a submitted job.

4.3. Interprocess communication:

PPU program - Subsystem communication:

A subsystem may request a specific PP program to be loaded in a PP for as long as the subsystem is in central memory. Further interaction between the subsystem and the PPU program may take place through mailboxes in the subsystem field length.

Subsystem - Subsystem communications:

Each subsystem may contain two buffers for messages in its field length. To send a message to subsystem B, subsystem A issues an SIC request indicating message buffer indices for A and B's name. When A issues an SIC request, the contents of A's message buffer are copied to B's message buffer. (by the master PPU)

A similar operation is available for B to get a message from A. In this case, B issues an RCB request, indicating buffer numbers in A and B, and A's name. When B issues an RCB request, the contents of A's message buffer are copied to B's buffer.

Both SIC and RCB can be performed with or without auto-recall.

5. Discussion

A system like KRONOS does not provide complete facilities for tasking or interprocess communication. It only provides some mechanisms that can be used to implement tasking or interprocess communication.

Because they are only simple mechanisms, they are easy to implement, they are fast in execution, and they suit most applications. The only problem with these is that the user has to devise policies and conventions on how to use them, and the user program includes a significant overhead to implement these policies.

On the other hand, if a system provides complete facilities for tasking or interprocess communications, these facilities have to imbed more elaborate conventions on their use. They would be more complicated, slower, and less "general-purpose". However, applications for which these facilities are well suited are easier to implement and do not include the same overhead.

All in all, both approaches may yield the same throughput or response time when considering Operating systems and application programs together. But the potential for enhancement appears greater in the second case.

In practice, it is difficult to look at operating systems from such a simple point of view.

1) It is difficult to make the distinction between what is called above a "mechanism" and a "complete service". Obviously that difference exists between interprocess communication in KRONOS and interprocess communication in MULTICS. There is however no sound criterion to make that distinction.
2) Facilities like tasking or interprocess communication are not independant from other aspects of operating systems (store management, protection, etc.)and result from complex design decisions.

However, this way of looking at operating systems may be practical to decide what to do next. I suggest

1)A study (in fact most of the work has been already done) of possible hardware enhancement of basic mechanisms like:

- interrupt mechanisms.
- context swapping mechanisms.
- scheduling primitives.
- programmed requests mechanisms.
- basic techniques for interprocess communications
- tasking mechanisms.

in various contexts (virtual memory, multiprocessor organizations, or simple PDP-11)

2)The harware enhancement of complete services in the context of a well defined operating systems; e.g. take a standard environment (some revision of the previous design.), a base processor, and some well defined goal for the operating system (e.g. multiprogramming, segmented memory, minimize response time), and propose a fast implementation , drawing upon the proposals of 1) above.

Primitive Functions

The primitive functions of an operating system can be roughly decomposed into:

1. Process (or task)management.
2. Processor management.
3. Memory management.
4. I/O Management.
5. Exception handling.

This decomposition is for prsctical purposes only, for choices made in one domain may seriously restrict the alternatives in another domain. This document attempts to make explicit the various choices made when selecting the primitives of our basic operating system, and the interplay of these design decision.

The goals of this design are indicated in the shopping list of table 1. In fellowing section, the five areas above are reviewed. For each area, the various alternatives of each design choice are explicited.

1.Process Management:

1.1 Process Definition:

The first choice is whether the system recognizes the set of applications as one single program (e.g. RT-11 without background job) or many entities like tasks or processes (ports in Umass, Jobs in SOS, tasks in the Representative Design, jobs in KRONOS). We choose the later.

Since we decide on managing a certain number of processes, a few

SHOPPING list:

This operating system is for the following sort of applications:

__ Time-sharing,
__ Batch Processing,
✓ Real Time Applications,
__ General Purpose.

Main objective (parameter that can be measured) is:

__ Minimize response time to TS users.
✓ Minimize response time of active processes to external events.
__ Maximize processor utilization.
__ Maximize memory utilization.
__ Maximize peripheral equipment utilization.

Secondary Objective:

__ Minimize response time to TS users.
__ Minimize response time of active processes to external events.
✓ Maximize processor utilization.
__ Maximize memory utilization.
__ Maximize peripheral equipment utilization.

Operating requirements (qualitative parameters) are:

__ Reliability. If so, indicate potentially adverse agents, e.g. user programs or external events:
__ Protection. If so, indicate what should be protected and against what:
✓ Cost efficiency.
__ Ease of modification of system and applications.

TABLE 1

questions arise:

a. What is a process ?

b. How is it recognized by the system ?

c. How many processes are managed?

Answering these questions already has a serious effect on Processor and Memory management techniques.

For simplicity, we describe a process as a collection of a main program along with some subroutines and some blocks of data. This correspond to the notion of a task in OS 360, program in Kronos, job in RT-11, and in SOS and the task representative Design.

The information the system maintains about a process contains at least:

- where it is located in primary or secondary storage.
- its current state (ready, active, blocked, etc)
- a "save area" holding the current values of registers when the process is interrupted.
- a list of the ressources the process owns or is entitled to use.
- a list of exception handling actions.
- additional information related to interprocess communications and synchronization.

The question of the number of processes is fairly complex in its implications. On one hand we could have a fixed number of processes which exists all the time (SOS, RT-11). On the other hand, we could have a variable number of processes (with may be a maximum number fixed by the system). This is the case of KRONOS and Umass (if one consider active ports

only). In the later case, we would have to decide how processes are created and destroyed, who may create and destroy processes, and what is the relationship between a craeted process and its creator.

The choice between fixed and variable number of processes depends also on whether a process must be a primary memory or can be swapped to secondary and remain "active". This in turn depends on whether processor management deals short term scheduling only (e.g. SOS) or with short and medium term scheduling (e.g. KRONOS).

For simplicity, let us decide that processes must be in primary memory. Two alternate solutions are:

a) Fixed number of processes: All processes are defined at system initialization. No creation or deletion of processes may occur during operation.

b) Variable number of processes: A certain number of processes are initialized when the system is bootstrapped (e.g. process accepting commands from the operator console). These processes can in turn create some processes and start them. No hierarchical ordering is imposed between a creating process and a created process. A process can terminate or be destroyed by another process. The primitives needed in this case are:

- create a process
- terminate a process

Because our basic operating system is aimed to manage a fairly static set of real time applications like RT-11 or SOS, the first solution is preferred. A fixed number of tasks is managed. All tasks are known at system generation and must be resident in primary memory.

1.2 Process Synchronization:

Process synchronization denotes the set of primitives by which a process can check the current status of other processes or can delay its execution until other processes have reached a certain state.

In RT-11 and Kronos (auto-recall feature), a process can synchronize itself only with a single process. The kind of event on which a process can synchronize itself can be a date (check whether the clock process has reached a certain date, or wait until a certain date occurs or until a certain amount of time has elapsed), or the completion of a given I/O operation. The corresponding primitives are:

- check the status of an event.
- wait for an event to occur.

In addition, RT-11 and SOS supply *SUSPEND* and *RESUME* requests by which a process can go to sleep and be later reactivated by another process. This equivalent to a binary semaphore on which signal and wait operations can be issued. Whereas the previous primitives "check" and "wait" allowed a regular process to synchronize itself with a pseudo process running under interrupt level, semaphores facilitate the synchronization of regular processes according to preestablished strategies. To facilitate the synchronization of more than two processes, general

semaphores are preferable to binary semaphores. In our operating system, semaphores are defined statically, at system generation time. The primitive operations on semaphores are the classic:

- signal (semaphore)
- wait (semaphore)

1.3 Inter-Process Communications.

Inter-Process Communication (IPC) denotes the primitives by which two processes can exchange short messages (e.g. application process and operator console process). RT-11 and KRONOS provide such facilities. There are two ways to consider IPC primitives:

1) IPC primitives are considered independently of other synchronization primitives as in RT-11 or KRONOS. IPC primitives include a connection primitive to establish a unidirectional communication link, send and receive operations similar to I/O read and write, and an acknowledgment protocol.

2) IPC operations are considered as an outgrowth of synchronization primitives. When two processes want to communicate, a mailbox is provided by the system. The two processes manage this mailbox according to a given strategy and coordinate their operations with semaphores (see Brinch Hansen p 104)

The first approach is needed in systems like KRONOS where processes cannot share a portion of the memory. The overhead of this method is rather high because messages and acknowledgments must be copied back and forth.

To keep a low overhead, and allow more flexibility at the process level, we choose the second solution, although it does not appear in any of the systems reviewed. It simplifies the internal structure of our system, but it also implies that processes should be allowed to share a message buffer.

Because of the applications projected for this system, message buffers are defined statically at sytem generation time and are resident in memory during system operations.

1.4 Process Management Implementation:

A process as defined in the previous sections can be in one of 3 states: blocked, ready, and running.

Blocked processes can either be waiting for an event or stopped at a semaphore. With each event is associated a queue. The basic operations needed to manage an event queue are:

- Enter a process into an event queue.
- Transfer all process from an event queue to the ready list.

With each semaphore is associated a counter and a queue. The basic operations needed to manage a semaphore are:

- Test semaphore counter.
- Increment, decrement semaphore counter.
- Insert a process at the end of the queue. } FIFO
- Remove a process from the beginning of the queue. }

2. Processor Management.

Processor Management denotes the set of operations and policies used to share the physical processor(s) between the pseudo processes and the regular processes.

Pseudo processes are interrupt servicing routines and run under hardware interrupt level. The system may manage information about these processes but their scheduling is usually imbedded in the base hardware (cf. PDP-11).

The regular processes are not directly activated by external events. Processes that are not blocked are either running or ready to run. The ready list contains the names of those processes that are eligible to run.

The scheduling algorithm used in Umass can be likened to a Round Robin algorithm. A sigle list of ready processes (compilations and executions of user programs) is managed in FIFO order. The processor is allocated for a time slice to the process on top of the list. If the process completes its time slice without blocking itself, it is returned to the end of the ready list.

In SOS, the list of ready processes is ordered by priority. The highest priority job is given the control of the CPU until it blocks itself or terminates executing the current task. If the process does not block itself, it is reinserted into the ready list according to its priority. An executing process may change its priority.

A process is allocated the processor for the duration of a whole task; there is no fixed time slice.

KRONOS uses a priority queue to allocate the processor between the processes resident in primary memory. The priority of a process can be fixed (high priority processes like TELEX), or variable. In the later case, the priority of a process depends on the base priority of the user and the anticipated use of ressources. The highest priority job on the ready list is given control of the CP for a fixed time slice. It is then reinserted into the ready list just before processes of lower priority.

For a real time operating system, the primary goal of the scheduler is to insure a minimum response time to external and internal events. There are two ways to look at this goal. This, in turn,suggests two different scheduling techniques.

1) To insure a minimum response time to a single event, the processor should be allocated to the process that deals with the event until this process terminates. Since many events may occur quasi-simultaneously, the various processes are assigned a priority. The scheduling algorithm uses a simple preemption scheme. When a process is dispatched to the ready queue, its priority is compared to the priority of the running process. If the running process has lower priority, it is preempted and returned to the ready list; the incoming process is allocated the processor. If the running process has a higher priority, the incoming process is inserted into the ready list. When a running process blocks itself or terminates, the highest priority process in the ready list is scheduled. To provide some flexibility, a running process may modify its

priority. If a running process lowers its priority, the ready list is inspected and the running process may be preempted to be replaced by the currently highest priority process. With this algorithm, there is no need to distinguish between hardware interrupt routines and other processes.

2) To insure a minimum response time to all events, in the average, all processes in the ready list should be allowed to advance smoothly. This implies a Round Robin scheme, where all processes have similar priority. Each ready process receive a fixed time slice on the processor before being returned to the end of the FIFO queue. Preemption is avoided. This scheme minimizes the system overhead.

There is no reason to prefer one scheme to the other. Queuing models do show that either one can be better than the other depending on the job mix. But these models rarely account for the CP time taken by the scheduler itself or by the preemtion mechanism. No choice will be made at this point and both techniques will be investigated further.

The various aspects of processor scheduling that are serious candidates for hardware enhancement are context swapping and preemption mechanisms, the ready queue and its manipulation primitives.

3. Memory Management.

Memory management denotes the set of techniques by which the physical memory is shared between the various processes.

For a real time system whose load is known at system generation time, little more is needed than what is provided by RT-11 or SOS. The memory space is allocated statically between the various processes when the system is generated. No protection between the various processes is supplied (the programmer is implicitely trusted, or the software development tools prevent undesirable interaction between processes).

The main reason for this approach is that real time processes must

be memory resident to provide acceptable response time. For those processes that are not so critical,e.g periodic checkpoints or processes that need to be scheduled every few minutes or so, it may be too expensive to provide enough memory. These processes can be decomposed into a resident root and one or many overlays. These processes may share (through semaphores) whatever space remains available (this space is known at system generation time) to load their overlays. The load point of these overlays is known at system genration time , so that no relation need be performed at load time. Semaphores and basic I/O operations are thus the only services required to manage overlays.

4. I/O Management:

By I/O management, we mean the techniques used to manage I/O devices, and the services provided to the user processes using these devices.

The first step is to specify what I/O devices are managed by the system.

RT-11 manages common peripherals only. Application programs may supply their handlers for private devices, but these handlers do not interface with the system (it is not possible to issue a WRITE request to a private device). In short RT-11 does not know about private devices. This is made possible by the vectored interrupt system of the PDP-11.

In all other systems we have reviewed, all devices (private and common devices) are known by the system. Each device has a handler (com-

posed of an initialization routine and an interrupt service routine) which is incorporated in the system at system generation time. In KRONOS, RT-11, and the representative Design, device handlers are not permanently resident. The only reason for doing so is to save memory for other use when some devices are not in use. For our design, we will assume that all device handlers are memory resident (first, memory is cheap; second, it might be impossible to predict that a device is not going to be used for a long period of time).

There are three types of I/O devices which are managed differently in most systems:

1. System owned/ system managed devices.
2. System owned / system allocated devices.
3. User owned / user managed devices.

The first two categories exists in all systems. System owned and managed devices are shared by the system and other processes, like a disk unit. These devices are file structured and can be used quasi simultaneously by many processes.

System owned and allocated devices are allocated by the system to a single user at a time (e.g. tape drive or plotter). A process must request the device before using it and return it to the system afterward.

User owned and managed devices are private devices that are dedicated to an application (e.g. radar, or dedicated terminal).

The various types of services provided by the system for each

type of device and the prospects for enhancement are discussed below.

4.1 System managed / system owned devices:

Most system owned and managed devices are mass storage devices, where the system maintains a file structure for itself and for user processes. The kind of services provided to the user includes:

- file management services: open file, close file, rename, change, purge, copy, etc.
- data transfer services: read a record from a file, write a record to a file.

These requests are processed by the file manager which, in turn, calls the device handler.

There is little prospect for the enhancement of most of these operations, because they depend on specific file structures and specific devices. There are, however two functions, that require our attention:

- file space allocation: Any system uses a track allocation algorithm (c.f. MST/TRT table and its management in KRONOS, and similar functions in Umass) which can be time consuming when the available file space is large.
- data transfer optimization: the average access time to a specific location on a mass storage device is very long. Studies show that when disk operations are performed in the order in which they are requested, the use of a moving head disk is far from optimium. Consequently, the average response time of I/O

bound processes is affected. By reordering the ~~requested~~ disk operations that are requested, the apparent access time to the disk can be lowered and a better response time can be obtained in the average.

4.2 System_owned_/_system allocated devices:

This type of device correspond to tape drives, plotter, etc, which can be accessed by one process at a time, for a long period of time (e.g. tape drive, paper tape punch, plotter). These devices must be explicitly allocated and returned. The services needed for this device include:

- allocate device.
- return device.

A process requesting the allocation of a device already allocated to another device is suspended until the device is returned to the system. There is no preemption. Furthermore, we will assume that no deadlock can occur in a real time system with a static set of applications programs.

The system owned and allocated devices do not usually have a file structure and the data transfer requests are processed entirely by the device handler. The data transfer requests are:

- Write data to device.
- Read data from device.

Because only one process can use the device for a period of time, there is no point optimizing the data transfers on these devices (which are most often sequential, anyhow). Only the allocation scheme may be a candi-

date for hardware enhancement.

4.3 User owned / user managed devices:

These devices are private devices whose handler is user supplied. These handlers follow system conventions so that user processes may request data transfer in a way similar to 4.2. The main difference with devices of 4.2 is that private devices are statically owned by some user processes.

5. Exception Handling.

Exception handling denotes the set of techniques by which exceptional events are detected and processed.

The first step is to define what an exception is. The initial concept can be best explained by a simple example. In most computers, a division by zero causes an internal interrupt. This interrupt is processed, in the usual fashion, and the interrupted program is either aborted or continued. This feature was implemented because it is faster than to force a program to check, before every division, that the divisor is non-zero, when such an operation is unlikely to occur. The gain in speed is obtained at the detection of the exceptional event, not during its processing.

In RT-11, the basic exceptions are illegal address (detected by time-out on the Unibus) and illegal instruction (detected in the central processing unit). In Umass, the exception are overflow, underflow, shift

faults, divide checks, and bound errors. In KRONOS, illegal instruction, address out of range, operand out of range, and indefinite operand are exceptions detected by the central processor.

Most operating systems define additional system dependent "exceptions". In KRONOS and RT-11, these exceptions correspond to illegal system requests, time limit, file limits, or file related operations that cannot be serviced. These exceptions are detected by software, rather than hardware. Their only resemblance with the previously defined exceptions is in the way they are processed. We choose not to include them in our design for the following reasons:

1) exceptions are considered only because of their interactions with other parts of a system. The hardware exceptions are sufficient from this point of view.
   Except for the way they are detected, system dependent exceptions are processed in the same way as hardware exceptions (System dependent exceptions do not require additional mechanism and would otherwise complicate our design. If it so happens that the hardware enhancement of our design provides for hardware detection of some system dependent exception, this exception will become undistinguishable from the initial harware exceptions.
2) The basic hardware exceptions may occur in a fully operating real time system, due to invalid external data (although a proper design would attempt to avoid them). System depen-

dent exceptions could only occur due to sloppy design or implementation. System dependent exceptions may be needed to debug application programs but they can represent an unnecessary overhead during full scale operation.

The set of exceptions in our system contains:

- Illegal address (unexisting physical address).
- Illegal instruction.
- Division by zero.
- Floating point exceptions: overflow, underflow, division by zero.

The next step is to define how exceptions are processed. In RT-11 and KRONOS, a user program can mask some of the hardware exceptions. In KRONOS , Umass, and RT-11, a user program may include a set of exception handling routines, These routnes are associated with the corresponding exception by a system request issued at the beginning of the program. In summary,there are three ways an exception can be handled:

1. Faulty process is aborted (unmasked exception).
2. Exception is ignored and faulty process is allowed to proceed (masked exception).
3. Control is given to an error handling routine supplied by the process (the content of the central registers at the time of the exception are passed to the exception handling routine).

These three possibilities are retained for our operating system. Because

of the static set of processes in operation, the type of action to be taken for each exception and for each process is defined statically at system generation time. Each process may have a different set of responses to the various exceptions. Each process has an exception table with a distinct entry for each exception. Each entry in that table contain either abort or mask (1 and 2 above) or the entry point of an exception routine (case 3 above).

Summary of Our Basic Operating System

This document summarizes the basic operating system abstracted from KRONOS, RT-11, SOS, TACFIRE, and the Representative design. A tentative set of system requests is included in the text.

1. General Overview.

Our basic operating system is intended for the run-time support of a fixed number of real time tasks. Its primary goal is to minimize the response time to external events.

A task or a process is composed of a main program and various subroutines. A process must be resident in primary memory. If a process has an overlay stucture, the main program or root of the process must be resident in primary memory. It is up to the root of a process to insure that the proper overlay is loaded before passing control to it.

The number of processes managed by the system is fixed at system generation time.

2. System request mechanism.

To request a service from the system, a process must first load one or two central registers with the appropriate parameters. In the case of data transfer related operations only, one of the parameters is a pointer to an area of memory called I/O descriptor.

The process then issues the appropriate service call instruction. Upon return from the system, the contents of the registers are generally undefined. In some cases, the systems returns one or more result values in the central registers.

3. Process Management.

3.1. Scheduling

Each process has a priority used for scheduling purposes. The highest priority ready process has control of the CPU. A running process may modify its current priority using the following request:

PRIORITY n , where n is between 0 and 128.

A running process that lowers its priority may be preemted.

3.2 Synchronization

.CHECK DATE

A process may check that the central clock has reached a certain date by issuing:CHECK_DATE date

Upon return, the central register R0 contains a value indicating whether the clock has a current value lower, equal or greater than the input parameter.

.Check I/O COMPLETION

A process may check whether an I/O operation formerly requested is still pending, has been initiated, or has completed by issuing:

CHECK_I/O I/O descriptor address

Upon return, teh central register R0 contains a value indicating the

current status of the designated I/O operation.

Note: If the system keeps the current status of the central clock in a fixed location, and records the status of an I/O operation in its descriptor, these requests can be eliminated (i.e. no service call needed; the program can check by itself, using a standard macro).

.WAIT DATE

A process can suspend itself until a certain date occurs by issuing

WAIT_DATE date , or

WAIT_PERIOD time

When the proper date occurs, or the amount of time is elapsed, the process is returned to the ready state. No parameter is returned by the system. process is resumed at next instruction following WAIT.

.WAIT_I/O_COMPLETION

A process can suspend itself until a formerly issued I/O operation has been completed, by issuing:

WAIT_I/O I/O descriptor address

No parameter is returned by the system.

.SEMAPHORES

General semaphores are available in our system. There are defined statically at system generation time. The available requests are:

SIGNAL_SEM semaphore #

WAIT_SEM semaphore #

When a SIGNAL_SEM request is issued, the first process of the semaphore queue is unblocked, or the semaphore counter is incremented if the queue is empty. A process issuing a SIGNAL_SEM request is preempted when it

unblocks a process of higher priority.

When a process issues a WAIT_SEM request, it is blocked and appended to the semaphore queue if the semaphore counter is zero. If the semaphore counter is positive, the process is allowed to proceed after decrementing the semaphore counter.

.INTERPROCESS COMMUNICATION.

Mailboxes are used for interprocess communications. MAilboxes are allocated statically at system generation time. Sending and receiving processes coordinate their use of a mailbox through semaphores.

4. Memory Management.

Most of the memory is allocated statically at system generation time. The remaining space is allocated dynamically to requesting processes for overlays and buffer space. The allocation is handled by the various processes using semaphores.

There is no hardware protection mechanism to protect one process from another at run time.

5.I/O Management.

5.1 File_management:

Processes can manipulate files located on system owned / system managed devices. File management requests are directed to the file manager. The requests include:

OPEN filename

CLOSE filename

PURGE filename

COPY filename

The requesting process is suspended until its request has been satisfied.

5.2. Device Management.

System owned / system allocated devices are allocated dynamically to requesting processes. Only one process at a time can own such a device. Requests are:

ALLOCATE_DEVICE device #

RETURN_DEVICE device #

A process requesting allocation of a device already allocated is suspended until the device becomes available. Requests are served in FIFO order. A process returning a device is preempted if the device is immediately allocated to a higher priority process.

5.3 Data_Transfers.

To issue a data transfer request, a process must first establish in memory an I/O transfer descriptor in a set of consecutive locations. An I/O transfer descriptor must contain:

- file operation or device operation indicator
- file name or device name
- record index or device address
- read / write
- location of buffer in memory
- length of buffer

- length of exchange (#of physical or logical records)
- a few words reserved for system control information and transfer status.

After loading R0 with the address of the I/O descriptor, the process issues one of the transfer requests below. The requests for file oriented transfers (system owned and managed devices) are directed to the file manager. The requests for device oriented transfers are directed to the device handler. Furthermore, a process may issue a transfer request and continue processing or issue a transfer request and suspend itself until completion of the transfer.

In the former case, the requests are:

READ_FILE	I/O descriptor address
WRITE_FILE	I/O descriptor address
READ_DEVICE	I/O descriptor address
WRITE_DEVICE	I/O descriptor address

It is possible for a process to check the status of a data transfer (CHECK_I/O) or to synchronize itself on the completion of that request (WAIT_I/O) after further processing.

In the later case, the requests are:

READ_FILE_WAIT	I/O descriptor address
WRITE_FILE_WAIT	" "
READ_DEVICE_WAIT	" "
WRITE_DEVICE_WAIT	" "

6.Exception Handling

The handling of each exception for each process is determined at system generation time. When an exception occurs, the system first deals with pending I/O. Depending on the exception table, the system will either:

a) Do nothing about I/O, or

b) Complete all pending I/O, or

c) Abort all pending I/O.

Then, the system will either continue the process, abort the process, or return control to an exception handling routine, as indicated in the exception table for this process. In the later case, the exception handling routine is passed the value of all registers at the time the exception occured, in a set of locations whose starting address is given in R0. The set of possible exception is:

- -Illegal address,
- - Illegal instruction,
- - Division by zero,
- - Floating point exceptions: overflow, underflow, and divison by zero.

7. Other facilities.

All the run time facilities supported by our system have been described above. Additional compile time facilities may be available, but are not considered in this ~~system~~ paper.

Part I: System Primitives

1. Processes

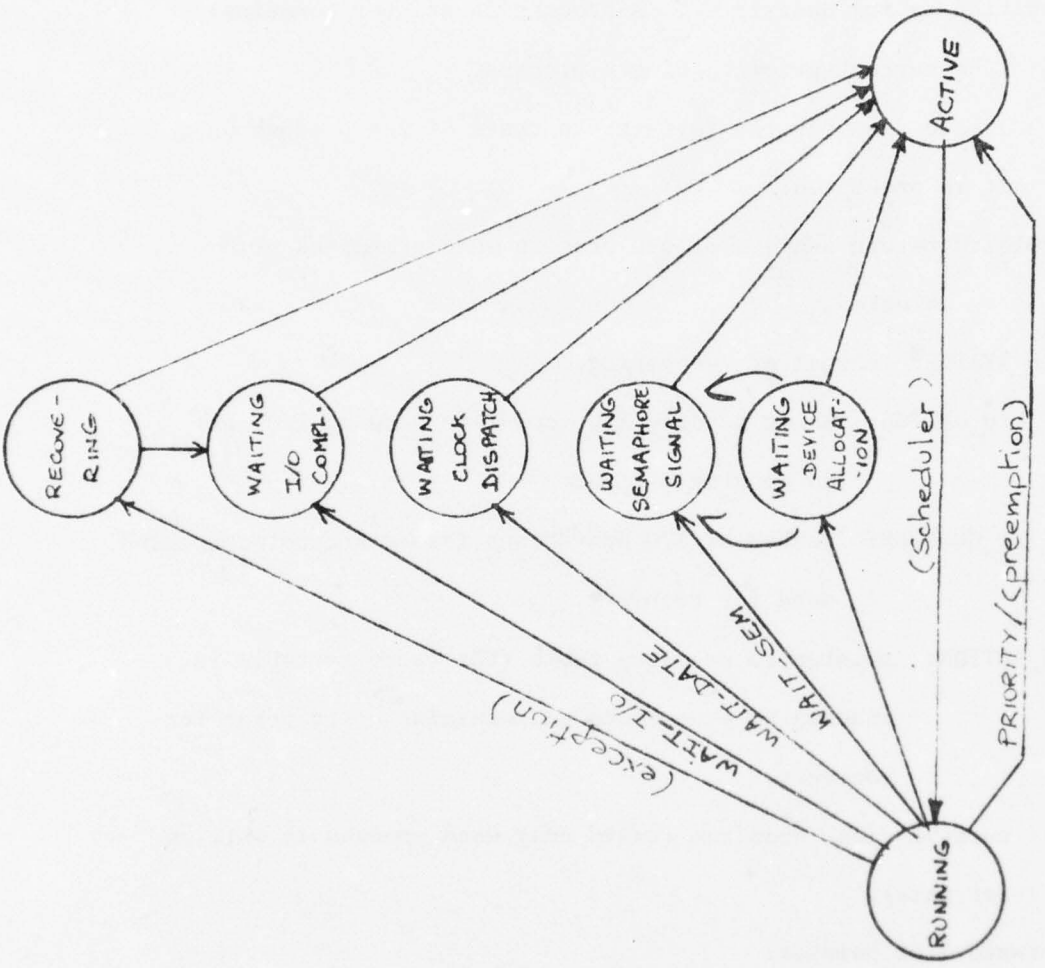
The states transition diagram of processes is given on page 2. Each process is identified by its descriptor. A process descriptor contains:

- PRIORITY: The current priority of the process.
- CONTEXT: a save area for the register contents of the process when it is preempted.
- RETURN_POINT: return address of the process when preempted or blocked.
- EXCEPTION_STATUS: normal or recovering.
- RECOVERY_I/O_OPTION: abort pending I/O, complete pending I/O, or do nothing.
- PENDING_I/O_COUNTER: number of I/O operations issued and not completed, used for recovery.
- RECOVERY_ACTION: pointer to recovery table (the recovery table is indexed by exceptions and contains entry point for recovery.
- -DEADLINE: current clock deadline (valid only when process is waiting for date).
- -NEXT: forward list pointer.

Except when running, a process belongs to one specific list (active list, semaphore queue, etc..). NEXT contains the name of the next process on the list or nil. When a process is running, the value of NEXT is irrelevant. The running process name is contained in the location CURRENT.

2. Active list.

This list is organized as a link list ordered by decreasing priority. All operations on this list are protected. The basic operations are insert,


RECOVE-RING
WAITING I/O COMPL.
WAITING CLOCK DISPATCH
WAITING SEMAPHORE SIGNAL
WAITING DEVICE ALLOCAT-ION
ACTIVE
RUNNING
(exception)
WAIT-I/O
WAIT-DATE
WAIT-SEM
(Scheduler)
PRIORITY (preemption)

remove, and compare priority of current running process with priority of the first active process. ACTIVE_FIRST and ACTIVE_LAST contain the name of the first and last processes on the active list, or nil if no process is active.

3. I/O Requests.

When issuing a READ or WRITE request, aprocess must specify the address of the descriptor of the operation. The descriptor contains:

- OWNER: the name of the process issuing the operation.
- DEVICE: the name of the device used for the operation.
- EXCHANGE: all data pertinent to the exchange (fixed size)
- I/O_STATUS: request pending, being performed, or completed.
- Next_DESCRIPTOR: The descriptor of a request which pending or being performed, always belongs to a list of requests associated with the device, NEXT_DESCRIPTOR contains the address of the next descriptor in this list, or nil.
- FIRST_WAITING_PROCESS: explained in 4.
- LAST_WAITING_PROCESS: explained in 4.
- TEMP: locations reserved for the file manager and the device handler.

The descriptors of the requests for an operation on device D are queued on a list associated with D. D_FIRST contains the address of the descriptor for the transfer being processed, or nil, if the list is empty (device inactive). D_LAST contains the address of the last descriptor on the list or nil. The list is usually managed in FIFO order, although some devices (e.g. disks) may have different strategies. Since descriptors are added to the list under program level and removed under interrupt level, all opeartions on the list must be protected.

4. I/O Synchronization.

Any process can suspend itself until completion of a transfer initiated by itself or any other process. When a process suspend itself, it is placed in a FIFO list associated with the transfer. Physically, the process descriptor is placed on a list associated with the descriptor

of the transfer. Each I/O descriptor contains two pointers FIRST_WAITING_PROCESS and LAST_WAITING_PROCESS used to manage the FIFO list of processes waiting for completion of the transfer. Processes are added to the list under program level. They are returned to the active list under interrupt level when the tranfer completes. A process attempting to block itself on a transfer already completed is returned to the active list immediately. Primitive operations on the synchronization list are enqueue and return queue to ACTIVE list.

5. Semaphores.

Each semaphore consists of a counter COUNT to record the number of SIGNAL's that have been received, and a (possibly empty) list of waiting processes managed with two pointers SEM_FIRST and SEM_LAST. The list of waiting processes is managed in FIFO order. Since Adding or removing processes to this list is caused by the running process, protected mode is required, to prevent preemption of the running process before completion of a SIGNAL or WAIT request.

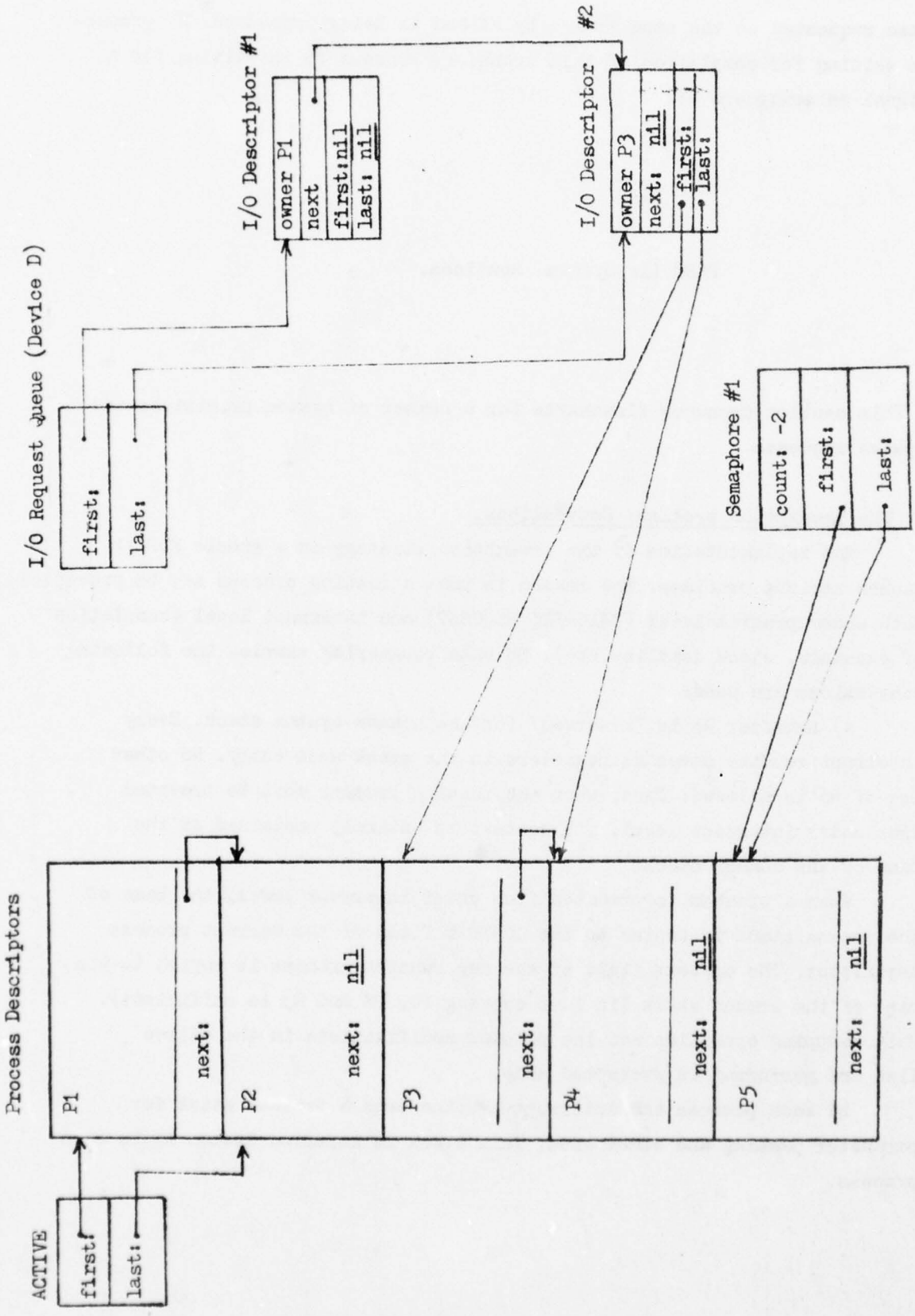
The mechanism for device allocation is equivalent to a binary semaphore and is implemented similarly.

6. Clock Synchronization.

Processes that suspend themselves until a certain date occurs are inserted into the Clock list. The real time clock process checks this list at every clock interrupt and returns to the active list processes whose deadline has expired. The list is managed by increasing deadline and uses two pointers CLOCK_FIRST and CLOCK_LAST. Since operations on the clock list are performed under interrupt level and program level, and may affect the running process, they must be performed in protected mode.

7. Example:

The illustration of page 5 shows a snapshot of a system with 6 processes. The running process P0 has been omitted from the illustration. Processes P1 and P2 are active. Processes P3 and P4 are waiting for the completion of a transfer requested by P3. Another transfer has previously


ACTIVE
first:
last:
Process Descriptors
P1
next:
P2
next: nil
P3
next:
P4
next: nil
P5
next: nil
I/O Request Queue (Device D)
first:
last:
I/O Descriptor #1
owner P1
next
first:nil
last: nil
I/O Descriptor #2
owner P3
next: nil
first:
last:
Semaphore #1
count: -2
first:
last:

been requested on the same device by P1 and is being processed. No process is waiting for completion of this transfer. Process P5 is waiting for a signal on semaphore #1.

PART II: System Routines.

This section presents flowcharts for a number of system primitives and system requests.

1. The preemption problem: Conventions.

The implementation of the preemption strategy on a simple PDP-11 causes serious problems. The reason is that a running process may be preempted both under program level (PRIORITY REQUEST) and interrupt level (completion of exchange, clock deadline,etc). To make bookeeping simple, the following conventions are used:

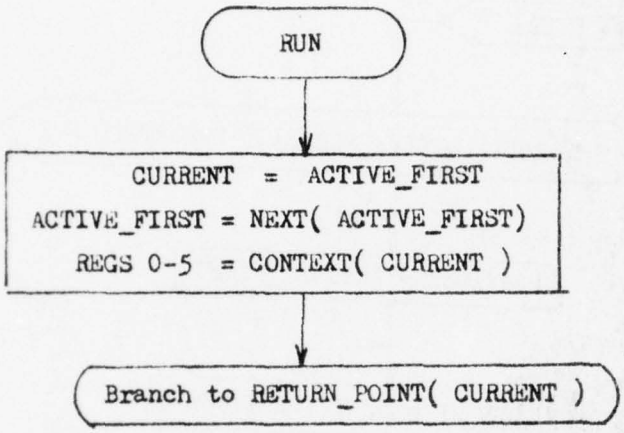
a) Register R6 is "reserved" for the common system stack. Every interrupt routine saves allregisters to the stack upon entry. No other use of R6 is allowed. Thus, when the running process must be preemted from under interrupt level, its context is entirely contained at the base of the common stack.

When a process is preemted from under interrupt level, the base of the common stack is copied to the CONTEXT field of the Current process descriptor. The context field of the new running process is copied to the base of the common stack (in fact copying PC, PS and R5 is sufficient). This swapping operation and the pointer modifications in the active list are performed in protected mode.

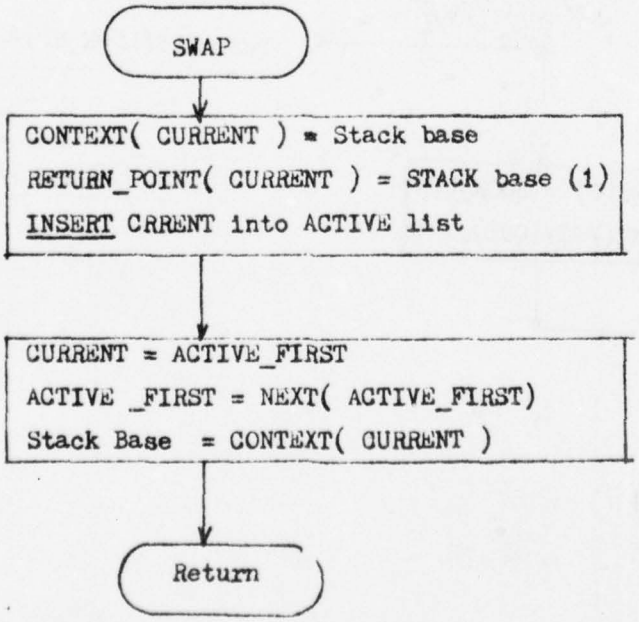
b) Each process and interrupt routine owns a private stack for paraneter passing and other uses, This stack is referred trough R5 in each process.

2. Active list Routines.

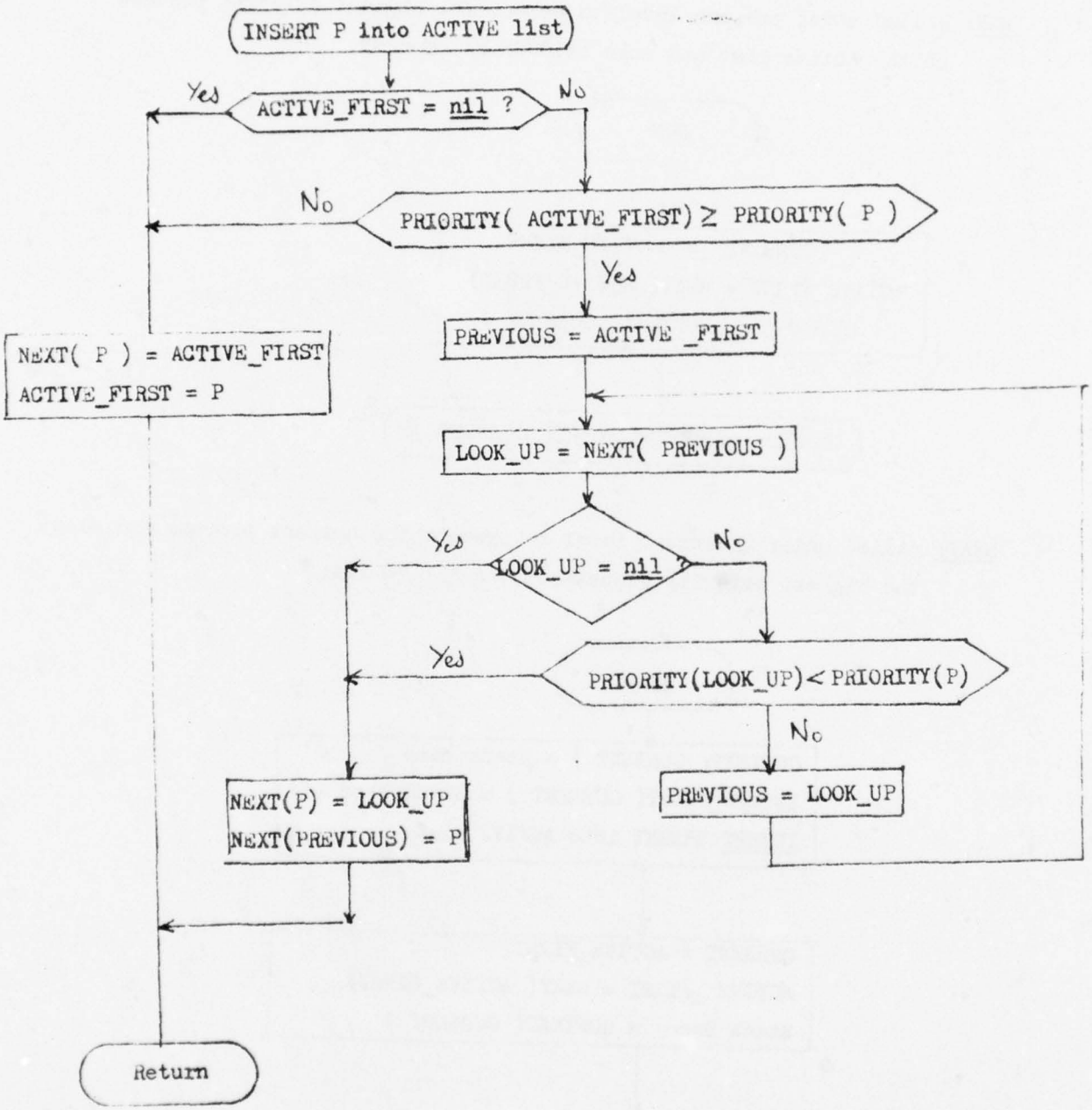
RUN: called under program level to select the highest priority process on the active list and make it CURRENT process.



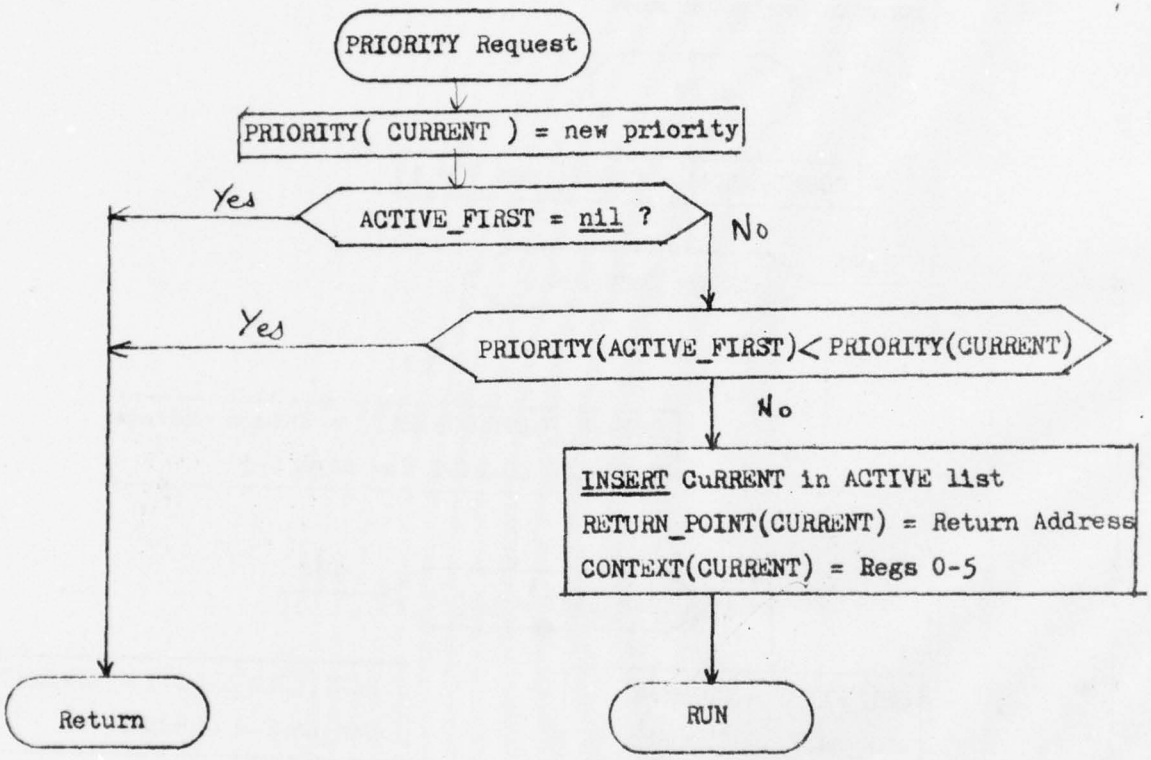
SWAP: called under interrupt level to preempt the current process and elect the highest priority process on the ACTIVE list.



INSERT: Called under any level to insert process P into the ACTIVE list.

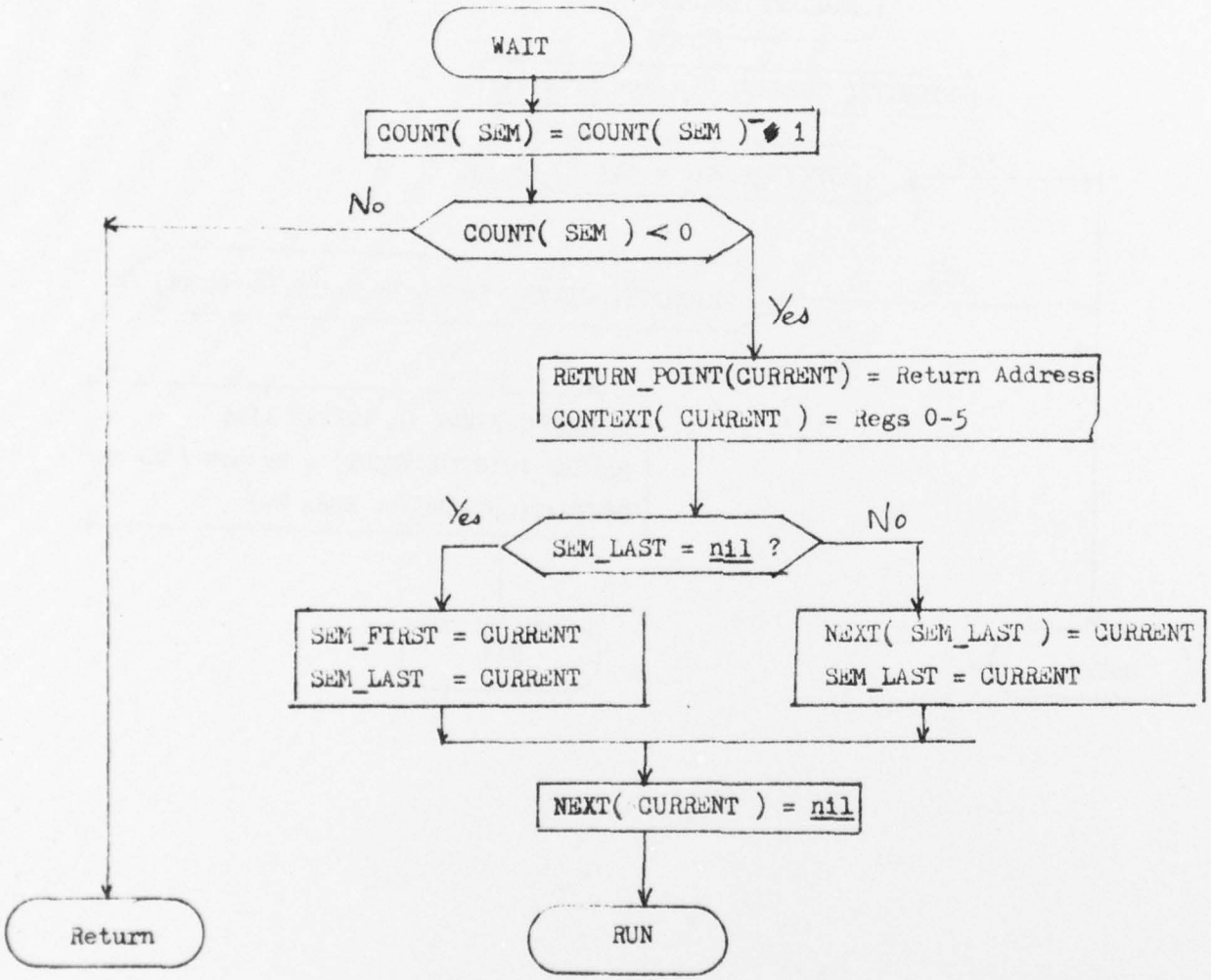


PRIORITY Request: - made by CURRENT process.
- may cause preemption of CURRENT process.
- runs under protected mode.

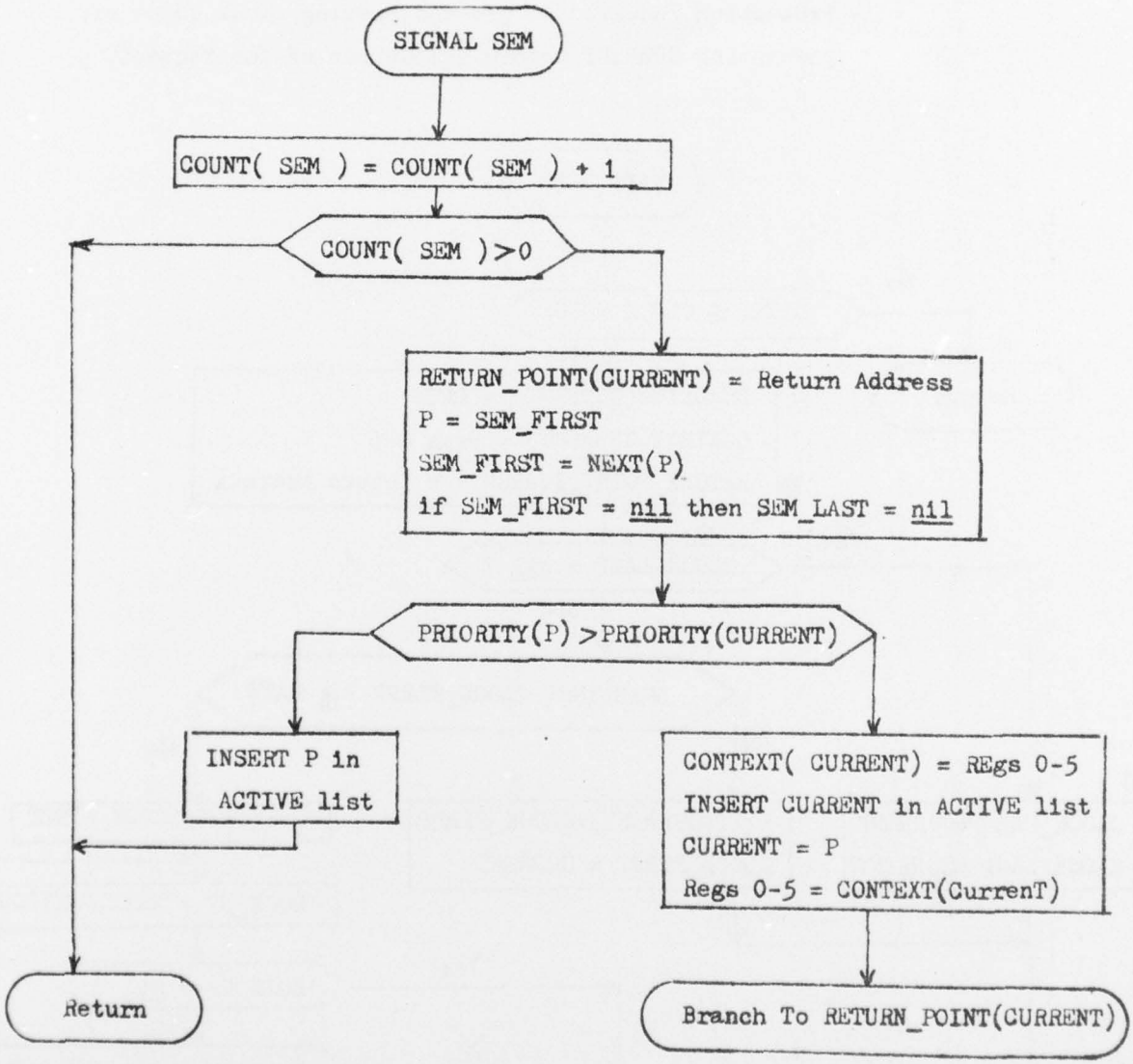


3. Semaphores.

WAIT SEM: - called by CURRENT under program level.
- may cause CURRENT to be suspended.
- runs in protected mode.

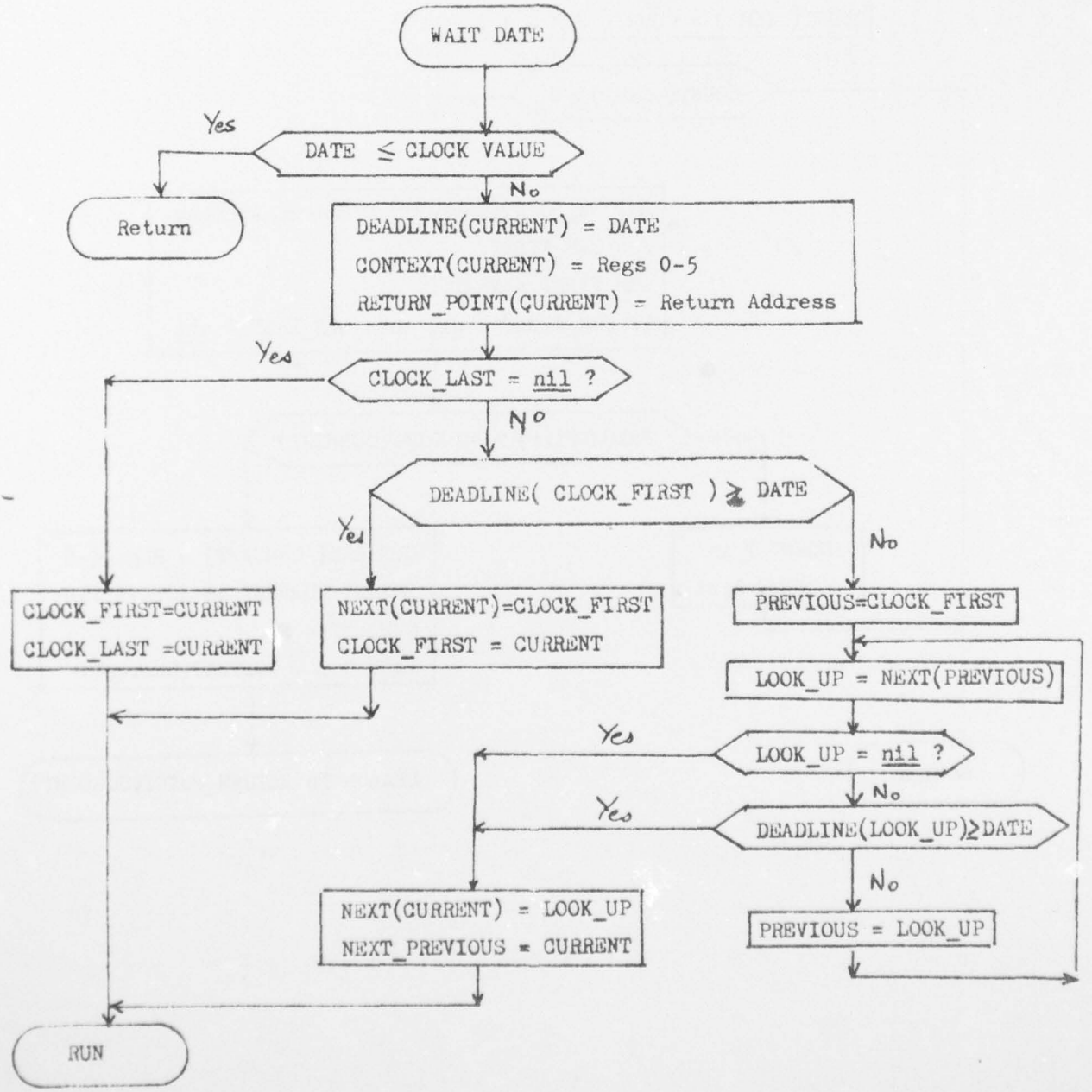


SIGNAL SEM: -called by CURRENT under program level.
- may cause CURRENT to be preempted.
- runs in protected mode.

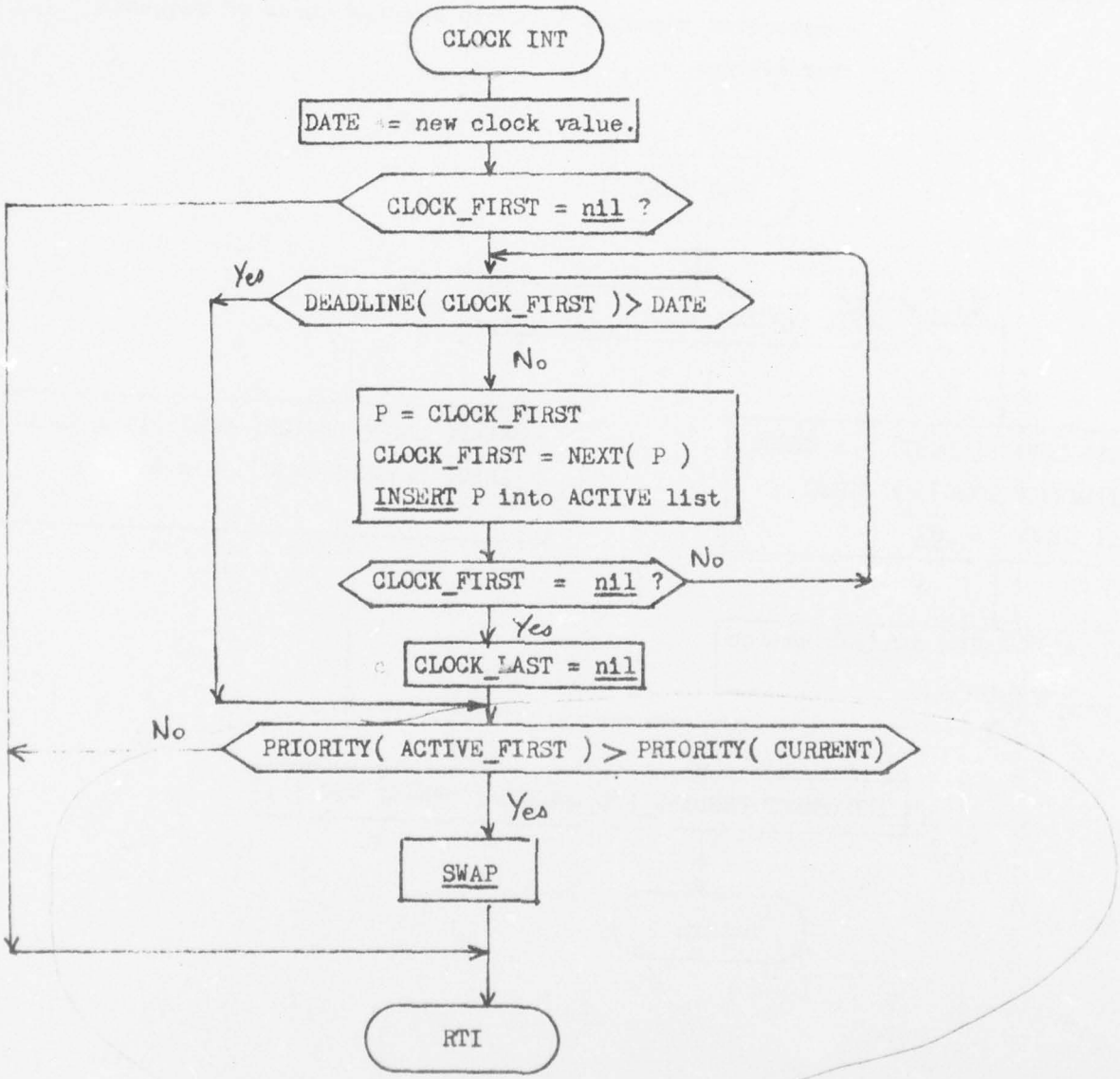


4. Clock Synchronization.

WAIT DATE: - called by CURRENT at user level. DATE is input parameter.
- CURRENT is inserted in Clock queue .
- Protection required to prevent messing clock queue or preempting CURRENT before completion of the request.

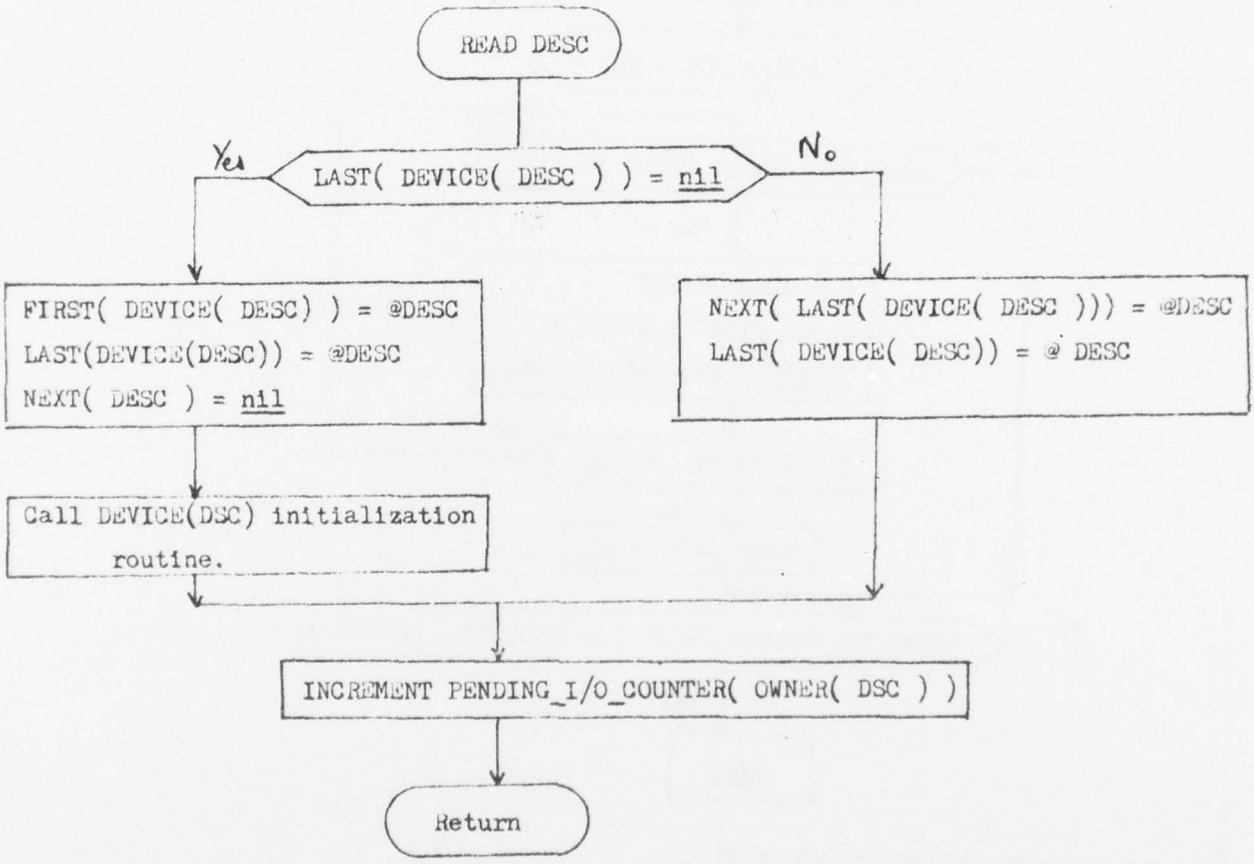


CLOCK Interrupt Routine: -Unprotected, runs under interrupt level.

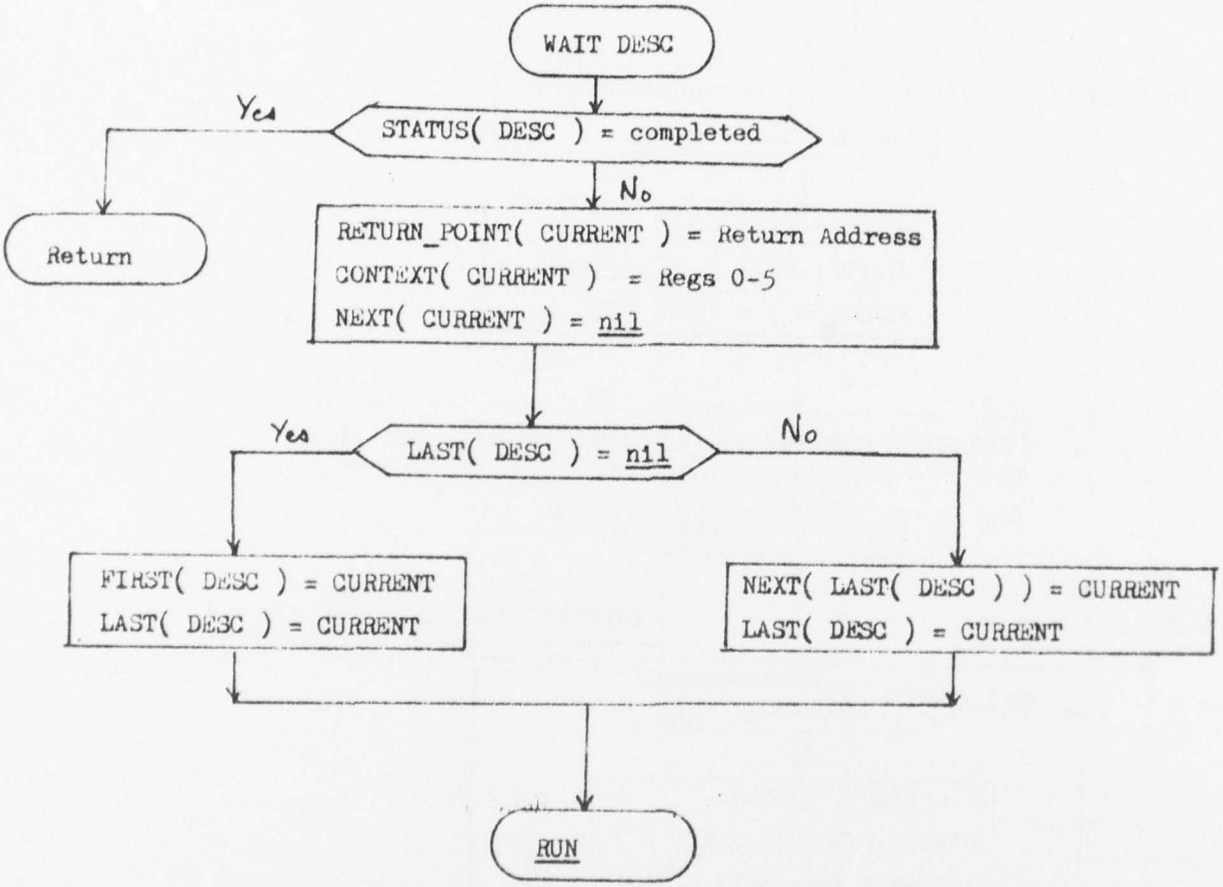


5. Input Output Transfers:

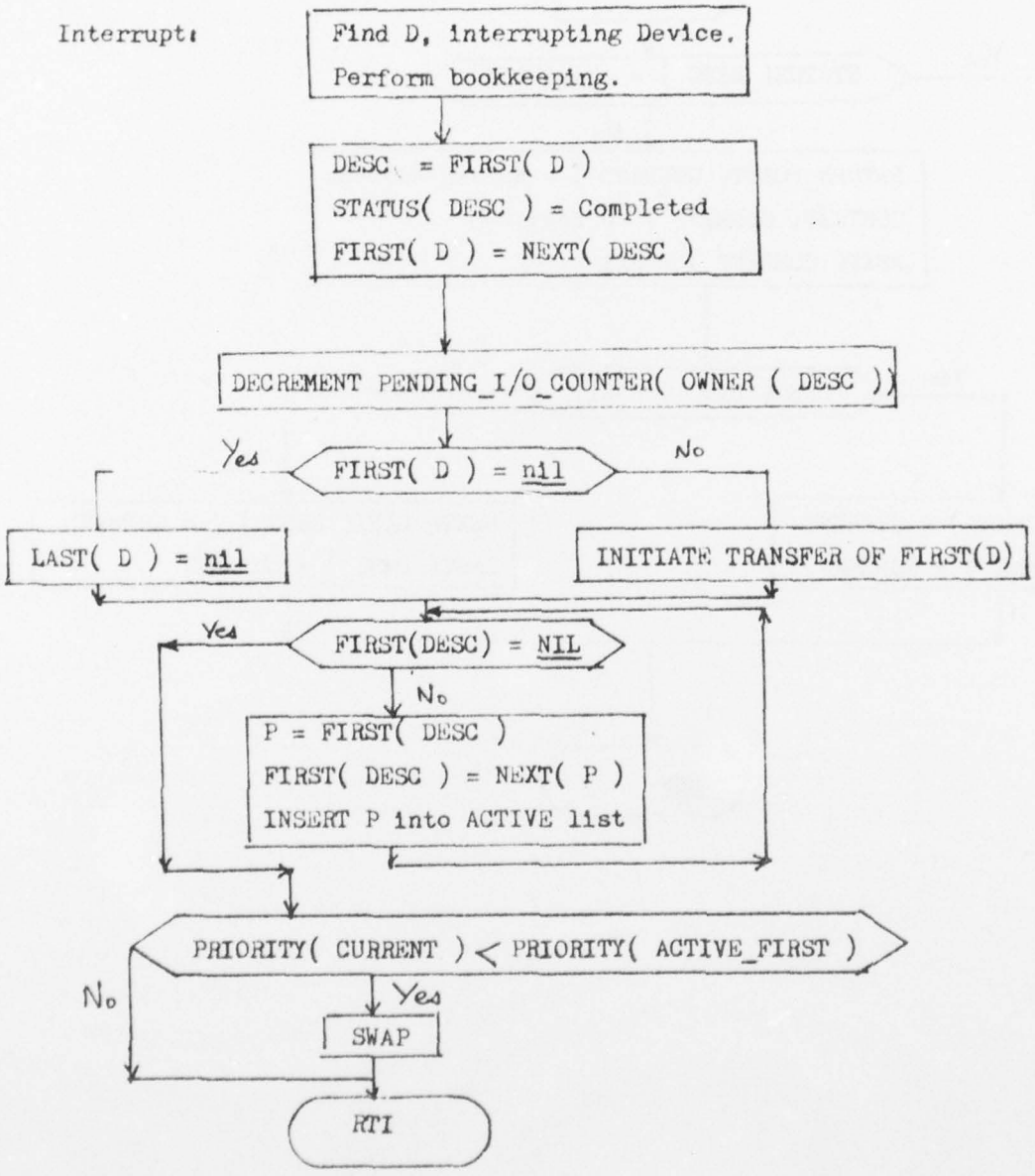
READ (I/O Descriptor): -Issued by CURRENT under program level.
-Protection required to avoid messing queue of requests for device.



WAIT I/O Descriptor: - Issued by CURRENT under program level.
- CURRENT process suspended and appended to list of waiting processes on I/O descriptor queue.
- operates under protected mode until new process elected from ACTIVE list (RUN).

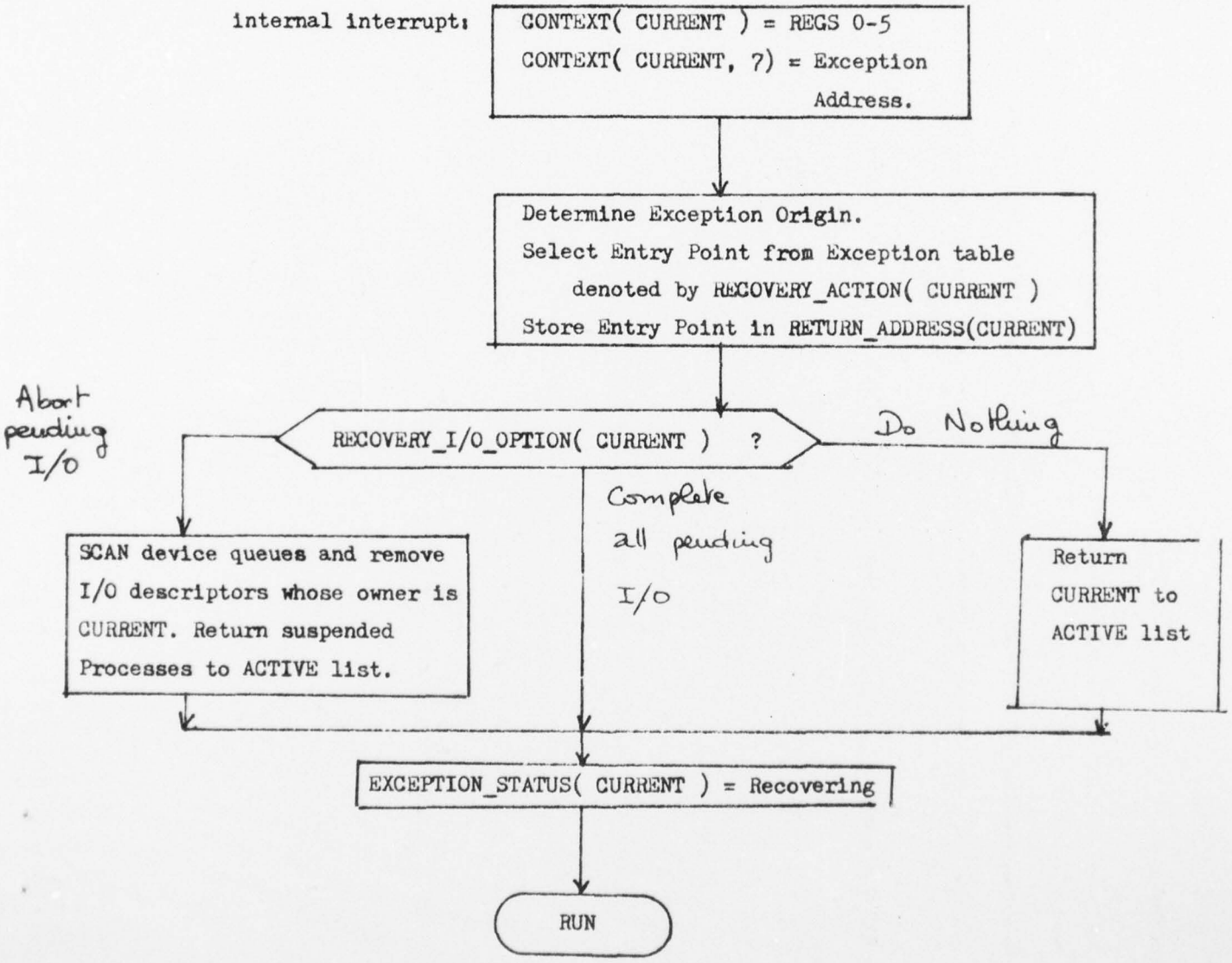


I/O handlers: - activated by interrupt on completion of transfer
- protected mode to prevend messing Device queues and ACTIVE queue.



6. Exception Handling.

Only the case of exceptions that create internal interrupts is presented. The I/o option requires modifications in the handlers presented in the previous section. These modifications are only discussed below.



I/O device Handlers must be modified so that, when the PENDING_I/O_COUNTER

of a Recovering process becomes zero, the process is returned to the ACTIVE list.

Synchronization

In any complex system with more than one processor, it is necessary to synchronize the various processes so that they do not interfere with each other. The classic examples of such synchronization are a producer-consumer relation and a race for a resource. There are several well-known solutions to the synchronization problem. Perhaps the best known are the P's and V's of Dijkstra. Other solutions include "message classes" and Petri nets. One of the simplest and most elegant is the "monitor" approach of Hoare. This method points out, quite reasonably, that if only one processor is "permitted" to do a thing, then there can't be any races involved. Let us look at these two classic examples and investigate how they are handled by both PV and by a monitor.

Consider a producer of messages and a consumer of those messages. Let there be a set of buffers between them to store messages that have been produced but not yet consumed. Now clearly the consumer can't take a message from an empty buffer nor should a producer try to add "just one more" if the buffer is already full.

We define three semaphores, "AVAIL" - the number of empty buffers, "FULL" - the number of full buffers, and "FLAG" - a mutual exclusion semaphore. We assume that P(S) will decrement S by one and continue if $S \geq 0$, and put the executer to sleep on S's queue if $S < 0$. V(S) will increment S by 1 and if $S \leq 0$ will wake up the process at the head of S's queue.

The producer executes the following operations:

```
producer: prepare message.
          P(AVAIL)
          P(FLAG)
          put message in buffer
          V(FULL)
          V(FLAG)
          GOTO produce
```

"nudge" its opposite number when it changes a flag. Notice that we have not had to appeal to any type of "non-interruptable operation" which lies hidden in the heart of each P and V.

Let us now examine briefly the critical section of a resource allocation algorithm. We have two or more processes that each have critical sections (RESOURCE 1, RESOURCE 2, etc.) which would interact detrimentally if they were allowed to execute concurrently. For example, the critical sections might involve obtaining a page of core. We bracket each critical section with a P and a V on some flag, thus allowing only one process to execute at a time within its critical section:

```
Process 1   P(FLAG)
            RESOURCE 1
            V(FLAG)
               .
               .

Process 2   P(FLAG)
            RESOURCE 2
            V(FLAG)
```

In this case the action is straightforward and relatively little confusion is possible, given that we assume that a P and a V are indivisible events. For most computers this involves an appeal to a non-interruptable instruction such as "replace add one" or "test and set".

To understand Hoare's monitor approach we need to realize that when a CPU is processing an interrupt it is behaving as an independent asynchronous processor, possibly racing with the main program for some resource. Each possible interrupt level constitutes another such "Pseudo-processor". Hoare shows that if we assign each critical section of code to exactly one processor or pseudo-processor, there can be no race conditions. Each process desiring a resource sets a flag requesting attention by the resource manager processor.

This may be recognized via polling by the manager or via an interrupt which creates an incorporation of the manager.

The manager, when awakened or "incorporated", examines the requests outstanding for its resource and arbitrates among them. Clearly this scheme will work if the manager is incorporated as the highest possible interrupt level and runs with interrupt off or if only one of the n different CPUs in a multiprocessor system ever executes the manager code. What Hoare points out is that it will work equally well with interrupt on (in the main program) provided only that none of the interrupt routines do any "messing about" with the resource. One might notice that there is very little, if any, differences in essence between a manager and an expanded "indivisible operation".

In our system we have chosen to use Hoare's monitor for the operating system itself, but we have provided P and V service calls for the benefit of those who do not understand monitors or who are dedicated disciples of Dijkstra.

The consumer does the following:

```
consumer: P(FULL)
          P(FLAG)
          get message from buffer
          V(AVAIL)
          V(FLAG)
          process message
          GOTO consume
```

The P's and V's on FLAG ensure that either the producer or the consumer is locked out and only one of them can be changing the buffer at a time. The producers P(AVAIL) will hang up if there are no buffers available and wait until the consumer does its V(AVAIL), freeing one up. Similarly, the consumers P(FULL) will cause the consumer to wait until a buffer has been filled before proceeding. The arrangement shown is taken from Tsichritzis and Bernstein and is correct as shown but as a measure of the intuitively non-obvious nature of these P's and V's answer the following question: "Will the algorithm still work properly if we reverse the order of the V(AVAIL) and V(FLAG) in the consumer?"

Now let us look at the way a monitor-type approach might handle the problem of producer consumer. The essence of this approach is that we avoid races by having only one contestant. Let there be a circular array of buffers and let each buffer have a one bit flag associated with it. If the flag is one the buffer contains a message. When the producer has a message ready to send, it looks for a buffer with a zero flag, writes the message in the buffer, and then sets the flag to one. Similarly the consumer looks for a full buffer, reads out the message and then clears the flag to zero. Obviously this will work and there is no question about the order in which things should happen. The only thing this description does not provide is for a "passive wait state" for the blocked consumer or producer. We can add this by having each process

Appendix

1. Although two papers were prepared for publication, neither has been accepted at this time.

2. Scientific personnel participating were
   Caxton C. Foster
   Maureen McCormack (earned MS)
   Steven Levitan
   Fredric Richard (earned PhD)